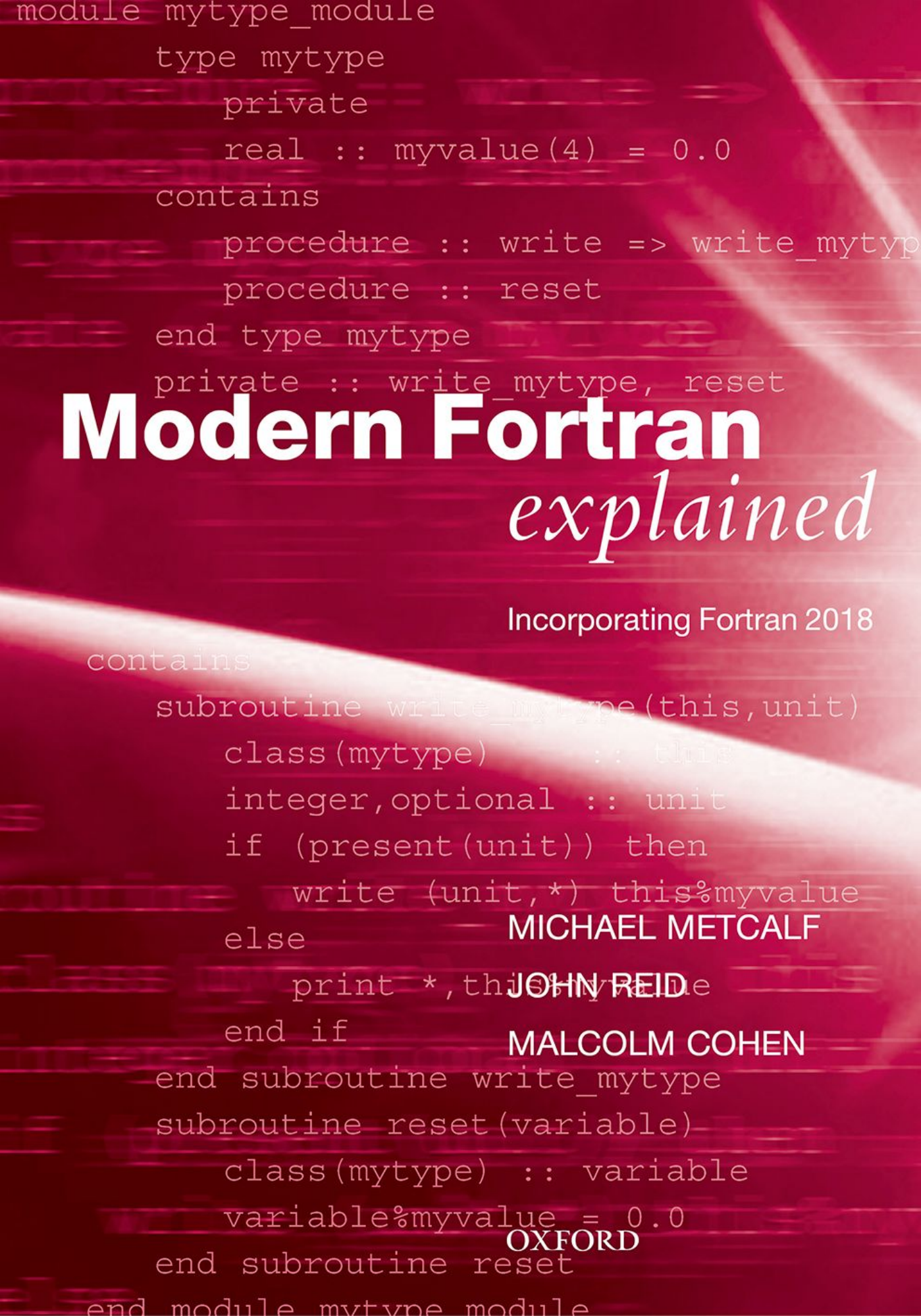


Modern Fortran *explained*

Incorporating Fortran 2018

MICHAEL METCALF

JOHN REID

MALCOLM COHEN

OXFORD

NUMERICAL MATHEMATICS AND SCIENTIFIC COMPUTATION

Series Editors

A. M. STUART E. SÜLI

NUMERICAL MATHEMATICS AND SCIENTIFIC COMPUTATION

Books in the series

Monographs marked with an asterisk (*) appeared in the series 'Monographs in Numerical Analysis' which is continued by the current series.

For a full list of titles please visit
https://global.oup.com/academic/content/series/n/numerical-mathematics-and-scientific-computation-nmsc/?lang=en&cc=cn

* J. H. Wilkinson: *The Algebraic Eigenvalue Problem*
* I. Duff, A. Erisman, and J. Reid: *Direct Methods for Sparse Matrices*
* M. J. Baines: *Moving Finite Elements*
* J. D. Pryce: *Numerical Solution of Sturm–Liouville Problems*

C. Schwab: *p- and hp- Finite Element Methods: Theory and Applications in Solid and Fluid Mechanics*
J. W. Jerome: *Modelling and Computation for Applications in Mathematics, Science, and Engineering*
A. Quarteroni and A. Valli: *Domain Decomposition Methods for Partial Differential Equations*
G. Em Karniadakis and S. J. Sherwin: *Spectral/hp Element Methods for Computational Fluid Dynamics*
I. Babuška and T. Strouboulis: *The Finite Element Method and its Reliability*
B. Mohammadi and O. Pironneau: *Applied Shape Optimization for Fluids*
S. Succi: *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*
P. Monk: *Finite Element Methods for Maxwell's Equations*
A. Bellen and M. Zennaro: *Numerical Methods for Delay Differential Equations*
J. Modersitzki: *Numerical Methods for Image Registration*
M. Feistauer, J. Felcman, and I. Straškraba: *Mathematical and Computational Methods for Compressible Flow*
W. Gautschi: *Orthogonal Polynomials: Computation and Approximation*
M. K. Ng: *Iterative Methods for Toeplitz Systems*
M. Metcalf, J. Reid, and M. Cohen: *Fortran 95/2003 Explained*
G. Em Karniadakis and S. Sherwin: *Spectral/hp Element Methods for Computational Fluid Dynamics*, Second Edition
D. A. Bini, G. Latouche, and B. Meini: *Numerical Methods for Structured Markov Chains*
H. Elman, D. Silvester, and A. Wathen: *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics*
M. Chu and G. Golub: *Inverse Eigenvalue Problems: Theory, Algorithms, and Applications*
J.-F. Gerbeau, C. Le Bris, and T. Lelièvre: *Mathematical Methods for the Magnetohydrodynamics of Liquid Metals*
G. Allaire and A. Craig: *Numerical Analysis and Optimization: An Introduction to Mathematical Modelling and Numerical Simulation*
K. Urban: *Wavelet Methods for Elliptic Partial Differential Equations*
B. Mohammadi and O. Pironneau: *Applied Shape Optimization for Fluids,* Second Edition
K. Boehmer: *Numerical Methods for Nonlinear Elliptic Differential Equations: A Synopsis*
M. Metcalf, J. Reid, and M. Cohen: *Modern Fortran Explained*
J. Liesen and Z. Strakoš: *Krylov Subspace Methods: Principles and Analysis*
R. Verfürth: *A Posteriori Error Estimation Techniques for Finite Element Methods*
H. Elman, D. Silvester, and A. Wathen: *Finite Elements and Fast Iterative Solvers: With Applications in Incompressible Fluid Dynamics*, Second Edition
I. Duff, A. Erisman, and J. Reid: *Direct Methods for Sparse Matrices*, Second Edition
M. Metcalf, J. Reid, and M. Cohen: *Modern Fortran Explained*, Second Edition

Modern Fortran Explained

Incorporating Fortran 2018

Michael Metcalf, John Reid, and Malcolm Cohen

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford, OX2 6DP,
United Kingdom

Oxford University Press is a department of the University of Oxford.
It furthers the University's objective of excellence in research, scholarship,
and education by publishing worldwide. Oxford is a registered trade mark of
Oxford University Press in the UK and in certain other countries

First edition published 1987 as *Fortran 8x Explained*
Second edition published 1989
Third edition published in 1990 as *Fortran 90 Explained*
Fourth edition published 1996 as *Fortran 90/95 Explained*
Fifth edition published 1999
Sixth edition published 2004 as *Fortran 95/2003 Explained*
Seventh edition published 2011 as *Modern Fortran Explained*
This edition published 2018

Impression: 1

Published in the United States of America by Oxford University Press
198 Madison Avenue, New York, NY 10016, United States of America

British Library Cataloguing in Publication Data

Data available

Library of Congress Control Number: 2018947662

ISBN 978–0–19–881189–3 (hbk.)
ISBN 978–0–19–881188–6 (pbk.)

DOI 10.1093/oso/9780198811886.001.0001

Printed and bound by
CPI Group (UK) Ltd, Croydon, CR0 4YY

Preface

Fortran remains one of the principal languages used in the fields of scientific, numerical, and engineering programming, and a series of revisions to the standard defining successive versions of the language has progressively enhanced its power and kept it competitive with several generations of rivals.

Beginning in 1978, the technical committee responsible for the development of Fortran standards, X3J3 (now PL22.3 but still informally called J3), laboured to produce a new, much-needed modern version of the language, Fortran 90. Its purpose was to 'promote portability, reliability, maintainability, and efficient execution ... on a variety of computing systems'. That standard was published in 1991, and work began in 1993 on a minor revision, known as Fortran 95. Subsequently, and with the same purpose, a further major upgrade to the language was prepared by J3 and the international committee, WG5. That revision, which included object-oriented programming features, is now known as Fortran 2003. This was followed by a further revision, Fortran 2008, including coarrays; and, most recently, a minor revision, Fortran 2018, including further coarray features. Once again, we have prepared a definitive informal description of the language that this latest standard defines. This continues the series of editions of this book – the two editions of *Fortran 8x Explained* that described the two drafts of the standard (1987 and 1989), *Fortran 90 Explained* that described the Fortran 90 standard (1990), two editions of *Fortran 90/95 Explained* that included Fortran 95 as well (1996 and 1999) and *Fortran 95/2003* (2004), with its added chapters on Fortran 2003. In that endeavour, a third co-author was welcomed. Finally, the first edition of *Modern Fortran Explained* (2011) added further chapters on Fortran 2008.

In this edition the basic language is Fortran 2008. An initial chapter sets out the background to the work on new standards, and Chapters 2 to 19 describe Fortran 2008 in a manner suitable both for grasping the implications of its features and for writing programs. The remaining chapters describe the additional features of Fortran 2018. Some knowledge of programming concepts is assumed. In order to reduce the number of forward references and also to enable, as quickly as possible, useful programs to be written based on material already absorbed, the order of presentation does not always follow that of the standard. In particular, we have chosen to defer to appendices the description of features that are officially labelled as redundant (some of which were deleted from the standard) and other features whose use we deprecate. They may be encountered in old programs, but are not needed in new ones.

Note that, apart from a small number of deletions, each of the languages Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, and Fortran 2018 is a subset of its successor.

In order to make the book a complete reference work it concludes with four appendices. They contain, successively, a description of various features whose use we deprecate and do not describe in the body of the book, a description of obsolescent and deleted features, an extended example illustrating the use of object orientation, and solutions to most of the exercises.

It is our hope that this book, by providing complete descriptions of Fortran 2008 and Fortran 2018, will continue the helpful role that earlier editions played for the corresponding versions of the standard, and that it will serve as a long-term reference work for the modern Fortran programming language.

Conventions used in this book

Fortran displayed text is set in typewriter font:

```
integer :: i, j
```

A line consisting of vertical dots (⋮):

```
subroutine sort
   ⋮
end subroutine sort
```

indicates omitted lines and an ellipsis (. . .):

```
 data_distance = ...
```

indicates omitted code.

Informal BNF terms are in italics:

`if (`*scalar-logical-expr*`)` *action-stmt*

Square brackets in italics indicate optional items:

`end if` *[name]*

and an ellipsis represents an arbitrary number of repeated items:

[`case` *selector [name]]*
block] ...

The italic letter *b* signifies a blank character.

Corrections to any significant errors detected in this book will be made available in the file *edits.pdf* at `ftp://ftp.numerical.rl.ac.uk/pub/MRandC`.

Contents

1. Whence Fortran?

1.1 Introduction

This book is concerned with the Fortran 2008 and Fortran 2018 programming languages, setting out a reasonably concise description of the whole language. The form chosen for its presentation is that of a textbook intended for use in teaching or learning the language.

After this introductory chapter, the chapters are written so that simple programs can be coded after the first three (on language elements, expressions and assignments, and control) have been read. Successively more complex programs can be written as the information in each subsequent chapter is absorbed. Chapter 5 describes the important concept of the module and the many aspects of procedures. Chapters 6 and 7 complete the description of the powerful array features, Chapter 8 considers the details of specifying data objects and derived types, and Chapter 9 details the intrinsic procedures. Chapters 10, 11 and 12 cover all the input/output features in a manner such that the reader can also approach this more difficult area feature by feature, but always with a useful subset already covered. Many, but not all, of the features described in Chapters 2 to 12 were available in Fortran 95.

Chapter 13 deals with parameterized data types, Chapter 14 with procedure pointers, Chapter 15 with object-oriented programming, and Chapter 16 with submodules. Coarrays, which are important for parallel processing, are described in Chapter 17. Chapter 18 describes floating-point exception handling, and Chapter 19 deals with interoperability with the C programming language. None of the features of Chapters 13 to 19 were available prior to Fortran 2003 or, in the cases of submodules and coarrays, Fortran 2008. Finally, the remaining Chapters, 20 to 23, describe the various enhancements brought to the language by the latest standard, Fortran 2018.

In Appendices A and B we describe features that are redundant in the language. Those of Appendix A are still fully part of the standard but their use is deprecated by us, while those of Appendix B are designated as obsolescent or deleted by the standard itself.

This introductory chapter has the task of setting the scene for those that follow. Section 1.2 presents the early history of Fortran, starting with its introduction over sixty years ago. Section 1.3 continues with the development of the Fortran 90 standard, summarizes its important new features, and outlines how standards are developed; Section 1.4 looks at the mechanism that has been adopted to permit the language to evolve. Sections 1.5 to 1.11 consider the development of Fortran 95 and its extensions, then of Fortran 2003, Fortran 2008, and Fortran 2018. The final section considers the requirements on programs and processors for conformance with the standard.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

1.2 Fortran's early history

Programming in the early days of computing was tedious in the extreme. Programmers required a detailed knowledge of the instructions, registers, and other aspects of the central processing unit (CPU) of the computer for which they were writing code. The **source code** itself was written in a numerical notation. In the course of time mnemonic codes were introduced, a form of coding known as **machine** or **assembly code**. These codes were translated into the instruction words by programs known as assemblers. In the 1950s it became increasingly apparent that this form of programming was highly inconvenient, although it did enable the CPU to be used in a very efficient way.

These difficulties spurred a team led by John Backus of IBM to develop one of the earliest high-level languages, Fortran. Their aim was to produce a language which would be simple to understand but almost as efficient in execution as assembly language. In this they succeeded beyond their wildest dreams. The language was indeed simple to learn, as it was possible to write mathematical formulae almost as they are usually written in mathematical texts. (The name Fortran is a contraction of 'formula translation'.) This enabled working programs to be written more quickly than before, for only a small loss in efficiency, as a great deal of care was devoted to the generation of fast object code.

But Fortran was revolutionary as well as innovatory. Programmers were relieved of the tedious burden of using assembler language, and computers became accessible to any scientist or engineer willing to devote a little effort to acquiring a working knowledge of Fortran. No longer was it necessary to be an expert on computers to be able to write application programs.

Fortran spread rapidly as it fulfilled a real need. Inevitably, dialects of the language developed, which led to problems in exchanging programs between computers, and so in 1966 the then American Standards Association (later the American National Standards Institute, ANSI) brought out the first ever standard for a programming language, now known as Fortran 66.

But the proliferation of dialects remained a problem after the publication of the 1966 standard. There was widespread implementation of features which were essential for large-scale programs, but which were ignored by the standard. Different compilers implemented such features in different ways. These difficulties were partially resolved by the publication of a new standard in 1978, known as Fortran 77, which included several new features that were based on vendor extensions or preprocessors.

1.3 The drive for the Fortran 90 standard

After thirty years' existence, Fortran was far from being the only programming language available on most computers, but Fortran's superiority had always been in the area of numerical, scientific, engineering, and technical applications and so, in order that it be brought properly up to date, the ANSI-accredited technical committee X3J3 (subsequently known as J3 and now formally as PL22.3), working as a development body for the ISO committee ISO/IEC JTC1/SC22/WG5, once again prepared a new standard, now known as Fortran 90. We will use the abbreviations J3 and WG5 for these two committees.

J3 is composed of representatives of computer hardware and software vendors, users, and academia. It is now accredited to NCITS (National Council for Information Technology Standards). J3 acts as the development body for the corresponding international group, WG5, consisting of international experts responsible for recommending the choice of features. J3 maintains other close contacts with the international community by welcoming foreign members, including the present authors over many years.

What were the justifications for continuing to revise the definition of the Fortran language? As well as standardizing vendor extensions, there was a need to modernize it in response to the developments in language design that had been exploited in other languages, such as APL, Algol 68, Pascal, Ada, C, and C++. Here, J3 could draw on the obvious benefits of concepts like data hiding. In the same vein was the need to begin to provide an alternative to dangerous storage association, to abolish the rigidity of the outmoded source form, and to improve further on the regularity of the language, as well as to increase the safety of programming in the language and to tighten the conformance requirements. To preserve the vast investment in Fortran 77 code, the whole of Fortran 77 was retained as a subset. However, unlike the previous standard, which resulted almost entirely from an effort to standardize existing practices, the Fortran 90 standard was much more a development of the language, introducing features which were new to Fortran but were based on experience in other languages.

The main features of Fortran 90 were, first and foremost, the array language and data abstraction. The former is built on whole-array operations and assignments, array sections, intrinsic procedures for arrays, and dynamic storage. It was designed with optimization in mind. Data abstraction is built on modules and module procedures, derived data types, operator overloading, and generic interfaces, together with pointers. Also important were the new facilities for numerical computation, including a set of numeric inquiry functions, the parameterization of the intrinsic types, new control constructs such as `select case` and new forms of `do`, internal and recursive procedures, optional and keyword arguments, improved I/O facilities, and many new intrinsic procedures. Last but not least were the new free source form, an improved style of attribute-oriented specifications, the `implicit none` statement, and a mechanism for identifying redundant features for subsequent removal from the language. The requirement on compilers to be able to identify, for example, syntax extensions, and to report why a program has been rejected, are also significant. The resulting language was not only a far more powerful tool than its predecessor, but a safer and more reliable one too. Storage association, with its attendant dangers, was not abolished, but rendered unnecessary. Indeed, experience showed that compilers detected errors far more frequently than before, resulting in a faster development cycle. The array syntax and recursion also allowed quite compact code to be written, a further aid to safe programming.

1.4 Language evolution

The procedures under which J3 works require that a period of notice be given before any existing feature is removed from the language. This means, in practice, a minimum of one revision cycle, which for Fortran is at least five years. The need to remove features is evident: if the only action of the committee is to add new features, the language will become

grotesquely large, with many overlapping and redundant items. The solution finally adopted by J3 was to publish as an appendix to a standard a set of two lists showing which items have been removed or are candidates for eventual removal.

One list contains the **deleted features**, those that have been removed. Since Fortran 90 contained the whole of Fortran 77, this list was empty for Fortran 90. It was also empty for Fortran 2008 but was not for the others, see Appendices B.4 to B.6.

The second list contains the **obsolescent features**, those considered to be outmoded and redundant, and which are candidates for deletion in the next revision. This list was empty for Fortran 2008 but was not for the others, see Appendices B.1 to B.3.

Apart from carriage control (Appendix B.5), the deleted features are still being supported by most compilers because of the demand for old tried and tested programs to continue to work. Thus, the concept of obsolescence is really not working as intended, but at least it gives a clear signal that certain features are outmoded, and should be avoided in new programs and not be taught to new programmers.

1.5 Fortran 95

Following the publication of the Fortran 90 standard in 1991, two further significant developments concerning the Fortran language occurred. The first was the continued operation of the two Fortran standards committees, J3 and WG5, and the second was the founding of the High Performance Fortran Forum (HPFF).

Early on in their deliberations, the standards committees decided on a strategy whereby a minor revision of Fortran 90 would be prepared by the mid-1990s and a major revision by about the year 2000. The first revision, Fortran 95, is the subject of the first part of this book.

The HPFF was set up in an effort to define a set of extensions to Fortran to make it possible to write portable code when using parallel computers for handling problems involving large sets of data that can be represented by regular grids. This version of Fortran was to be known as High Performance Fortran (HPF), and it was quickly decided, given the array features of Fortran 90, that it, and not Fortran 77, should be its base language. The final form of HPF[1] was a superset of Fortran 90, the main extensions being in the form of directives that take the form of Fortran 90 comment lines, and are thus recognized as directives only by an HPF processor. However, it also became necessary to add some additional syntax, as not all the desired features could be accommodated in the form of such directives.

The work of J3 and WG5 went on at the same time as that of HPFF, and the bodies liaised closely. It was evident that, in order to avoid the development of divergent dialects of Fortran, it would be desirable to include the new syntax defined by HPFF in Fortran 95 and, indeed, the HPF features were its most significant new features. Beyond this, a small number of other pressing but minor language changes were made, mainly based on experience with the use of Fortran 90.

The details of Fortran 95 were finalized in 1995, and the new ISO standard, replacing Fortran 90, was adopted in 1997, following successful ballots, as ISO/IEC 1539-1:1997.

[1] *The High Performance Fortran Handbook*, C. Koebel et al., MIT Press, Cambridge, MA, 1994.

1.6 Extensions to Fortran 95

Soon after the publication of Fortran 90, an auxiliary standard for varying-length strings was developed. A minority felt that this should have been part of Fortran 90, but were satisfied with this alternative. The auxiliary standard defines the interface and semantics for a module that provides facilities for the manipulation of character strings of arbitrary and dynamically variable length. It has been revised for Fortran 95 as ISO/IEC 1539-2:2000(E). An annex referenced a possible implementation[2] in Fortran 95, which demonstrated its feasibility. The intention was that vendors provide equivalent features that execute more efficiently but, in fact, that never happened.

Further, in 1995, WG5 decided that three features,

- permitting allocatable arrays as structure components, dummy arguments, and function results,
- handling floating-point exceptions, and
- interoperability with C,

were so urgently needed in Fortran that it established development bodies to develop 'Technical Reports of Type 2'. The intent was that the material of these technical reports be implemented as extensions of Fortran 95 and integrated into the next revision of the standard, apart from any defects found in the field. It was essentially a beta-test facility for a language feature. In the event, the first two reports were completed, but not the third. The first was widely implemented in Fortran 95 compilers. The features of the two reports were incorporated in Fortran 2003. While there was no report for the third, features for interoperability with C were included in Fortran 2003. In this book, the three topics are included in Chapters 6, 18, and 19, respectively.

1.7 Fortran 2003

The next full language revision was published in November 2004 and is known as Fortran 2003 since the details were finalized in 2003. Unlike Fortran 95, it was a major revision, its main new features being:

- Derived type enhancements: parameterized derived types, improved control of accessibility, improved structure constructors, and finalizers.
- Object-oriented programming support: type extension and inheritance, polymorphism, dynamic type allocation, and type-bound procedures.
- Data manipulation enhancements: allocatable components, deferred type parameters, `volatile` attribute, explicit type specification in array constructors and `allocate` statements, pointer enhancements, extended initialization expressions (now called constant expressions), and enhanced intrinsic procedures.

[2] `ftp://ftp.nag.co.uk/sc22wg5/ISO_VARYING_STRING/`

- Input/output enhancements: asynchronous transfer, stream access, user-specified transfer operations for derived types, user-specified control of rounding during format conversions, named constants for preconnected units, the `flush` statement, regularization of keywords, and access to error messages.
- Procedure pointers.
- Support of IEC 60559 (IEEE 754) exceptions.
- Interoperability with the C programming language.
- Support for international usage: access to ISO 10646 four-byte characters and choice of decimal point or comma in numeric formatted I/O.
- Enhanced integration with the host operating system: access to command line arguments, environment variables, and processor error messages.

1.8 Extensions to Fortran 2003

It was decided in 2001 that a feature to address problems with the maintenance of very large modules was too important to delay for the next revision and should be the subject of a further Technical Report. This was completed in 2005 as ISO/IEC TR 19767:2005 and defines submodules, see Chapter 16. Unfortunately, there were few early implementations.

The features for interoperability in Fortran 2003 provide a mechanism for sharing data between Fortran and C, but it was still necessary for users to implement a translation layer for a procedure with an argument that is optional, a pointer, allocatable, or of assumed shape. It was decided that adding features to address this was too important to wait for the next revision and so was made the subject of another Technical Report. Work began in 2006 but proved more difficult than expected and was not completed until 2012, so was not ready for incorporation in the next standard, Fortran 2008. It appeared as ISO/IEC TS 29113:2012 and is incorporated in Fortran 2018. ISO/IEC had meanwhile renamed these reports as Technical Specifications, but the intention was unchanged.

1.9 Fortran 2008

Notwithstanding the fact that compilers for Fortran 2003 were very slow to appear, the standardization committees thought fit to plunge on with another standard. It was intended to be a small revision, but it became apparent that parallel programming was going to be universal, so coarrays were added and were its single most important new feature. Further, the `do concurrent` form of loop control and the `contiguous` attribute were introduced. The submodule feature, promised by the Technical Report mentioned in Section 1.8, was added. Other new features include various data enhancements, enhanced access to data objects, enhancements to I/O and to execution control, and more intrinsic procedures, in particular for bit processing.

Fortran 2008 was published in October 2010.

1.10 Extensions to Fortran 2008

To avoid the extension of Fortran 2003 to Fortran 2008 becoming very large and to speed up agreement on all the details, WG5 decided to defer some of the coarray features to a Technical Specification. As for the Tecnnical Reports, it was intended that the material would be implemented as extensions of Fortran 2008 and integrated into the next revision, apart from any defects found in the field. Work did not start until 2011 and was completed in 2015 as ISO/IEC TS 18508:2015. The features are described in Chapter 20.

1.11 Fortran 2018

Fortran 2018 is a minor extension of Fortran 2008. The features were chosen in 2015 and the language was originally called Fortran 2015. It was finally decided to follow the more usual practice of naming the version by its date of publication; this gives the impression that the gap from the previous version is larger than it really is.

The main changes concern the features defined in the Technical Specifiactions for further coarray features, see Chapter 20, and further interoperability with C, see Chapter 21. Beyond these, there are quite a large number of small improvements, see Chapters 22 and 23.

1.12 Conformance

The standards are almost exclusively concerned with the rules for programs rather than for processors. A processor is required to accept a **standard-conforming program** and to interpret it according to the standard, subject to limits that the processor may impose on the size and complexity of the program. The processor is allowed to accept further syntax and to interpret relationships that are not specified in the standard, provided they do not conflict with the standard. In many places in this book we say '... is not permitted'. By this we mean that it is not permitted in a standard-conforming program. An implementation may nevertheless permit it as an extension. Of course, the programmer must avoid such syntax extensions if portability is desired.

The interpretation of some of the standard syntax is **processor dependent**, that is, may vary from processor to processor. For example, the set of characters allowed in character strings is processor dependent. Care must be taken whenever a processor-dependent feature is used in case it leads to the program not being portable to a desired processor.

A drawback of the Fortran 77 standard was that it made no statement about requiring processors to provide a means to detect any departure from the allowed syntax by a program, as long as that departure did not conflict with the syntax rules defined by the standard. The new standards are written in a different style. The syntax rules are expressed in a variant of BNF[3] with associated constraints, and the semantics are described by the text. This semi-formal style is not used in this book, so an example, from Fortran 95, is perhaps helpful:

[3] Backus-Naur form, a notation used to describe the syntax of a computer language.

R609	*substring*	**is**	*parent-string* (*substring-range*)
R610	*parent-string*	**is**	*scalar-variable-name*
		or	*array-element*
		or	*scalar-structure-component*
		or	*scalar-constant*
R611	*substring-range*	**is**	*[scalar-int-expr]* : *[scalar-int-expr]*

Constraint: *parent-string* shall be of type character.

The value of the first *scalar-int-expr* in *substring-range* is called the **starting point** and the value of the second one is called the **ending point**. The length of a substring is the number of characters in the substring and is $\mathrm{MAX}(\ell - f + 1, 0)$, where f and ℓ are the starting and ending points, respectively.

Here, the three production rules and the associated constraint for a character substring are defined, and the meaning of the length of such a substring explained.

The standards are written in such a way that a processor, at compilation time, may check that the program satisfies all the constraints. In particular, the processor must provide a capability to detect and report the use of any

- obsolescent feature;
- additional syntax;
- kind type parameter (Section 2.5) that it does not support;
- non-standard source form or character;
- name that is inconsistent with the scoping rules; or
- non-standard intrinsic procedure.

Furthermore, it must be able to report the reason for rejecting a program. These capabilities are of great value in producing correct and portable code.

2. Language elements

2.1 Introduction

Written prose in a natural language, such as an English text, is composed firstly of basic elements – the letters of the alphabet. These are combined into larger entities, words, which convey the basic concepts of objects, actions, and qualifications. The words of the language can be further combined into larger units, phrases and sentences, according to certain rules. One set of rules defines the grammar. This tells us whether a certain combination of words is correct in that it conforms to the **syntax** of the language; that is, those acknowledged forms which are regarded as correct renderings of the meanings we wish to express. Sentences can in turn be joined together into paragraphs, which conventionally contain the composite meaning of their constituent sentences, each paragraph expressing a larger unit of information. In a novel, sequences of paragraphs become chapters and the chapters together form a book, which usually is a self-contained work, largely independent of all other books.

2.2 Fortran character set

Analogies to these concepts are found in a programming language. In Fortran, the basic elements, or character set, are the 26 letters of the English alphabet, in both upper and lower case, the ten Arabic numerals, 0 to 9, the underscore, _, and the so-called special characters listed in Table 2.1. Within the Fortran syntax, the lower-case letters are equivalent to the corresponding upper-case letters; they are distinguished only when they form part of a character sequence. In this book, syntactically significant characters will always be written in lower case. The letters, numerals, and underscore are known as **alphanumeric** characters.

Except for the currency symbol, whose graphic may vary (for example, £ in the United Kingdom), the graphics are fixed, though their styles are not fixed. As shown in Table 2.1, some of the special characters have no specific meaning within the Fortran language.

In the course of this and the following chapters we shall see how further analogies with natural language may be drawn. The unit of Fortran information is the **lexical token**, which corresponds to a word or punctuation mark. Adjacent tokens are usually separated by spaces or the end of a line, but sensible exceptions are allowed just as for a punctuation mark in prose. Sequences of tokens form **statements**, corresponding to sentences. Statements, like sentences, may be joined to form larger units like paragraphs. In Fortran these are known as **program units**, and out of these may be built a **program**. A program forms a complete

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

Table 2.1. The special characters of the Fortran language.

	Syntactic meaning		Syntactic meaning		No syntactic meaning
=	Equals sign	:	Colon	\	Backslash
+	Plus sign		Blank	$	Currency symbol
-	Minus sign	!	Exclamation mark	?	Question mark
*	Asterisk	%	Percent	{	Left curly bracket
/	Slash	&	Ampersand	}	Right curly bracket
(	Left parenthesis	;	Semicolon	~	Tilde
)	Right parenthesis	<	Less than	`	Grave accent
[	Left square bracket	>	Greater than	^	Circumflex accent
]	Right square bracket	'	Apostrophe	|	Vertical line
,	Comma	"	Quotation mark	#	Number sign
.	Decimal point			@	Commercial at

set of instructions to a computer to carry out a defined sequence of operations. The simplest program may consist of only a few statements, but programs of more than 100 000 statements are now quite common.

2.3 Tokens

Within the context of Fortran, alphanumeric characters (the letters, the underscore, and the numerals) may be combined into sequences that have one or more meanings. For instance, one of the meanings of the sequence `999` is a constant in the mathematical sense. Similarly, the sequence `date` might represent, as one possible interpretation, a variable quantity to which we assign the calendar date.

The special characters are used to separate such sequences and also have various meanings. We shall see how the asterisk is used to specify the operation of multiplication, as in `x*y`, and also has a number of other interpretations.

Basic significant sequences of alphanumeric characters or of special characters are referred to as **tokens**; they are labels, keywords, names, constants (other than complex literal constants), **operators** (listed in Table 3.4, Section 3.9), and **separators**, which are

```
/   (   )   [   ]   (/   /)   ,   =   =>   :   ::   ;   %
```

For example, the expression `x*y` contains the three tokens `x`, `*`, and `y`.

Apart from within a character string or within a token, blanks may be used freely to improve the layout. Thus, whereas the variable `date` may not be written as `d a t e`, the sequence `x * y` is syntactically equivalent to `x*y`. In this context, multiple blanks are syntactically equivalent to a single blank.

A name, constant, or label must be separated from an adjacent keyword, name, constant, or label by one or more blanks or by the end of a line. For instance, in

```
      real x
      rewind 10
   30 do k=1,3
```

the blanks are required after `real`, `rewind`, `30`, and `do`. Likewise, adjacent keywords must normally be separated, but some pairs of keywords, such as `else if`, are not required to be separated. Similarly, some keywords may be split; for example, `inout` may be written `in out`. We do not use these alternatives, but the exact rules are given in Table 2.2.

Table 2.2. Adjacent keywords where separating blanks are optional.

`block data`	`double precision`	`else if`	`else where`
`end associate`	`end block`	`end block data`	`end critical`
`end do`	`end enum`	`end file`	`end forall`
`end function`	`end if`	`end interface`	`end module`
`end procedure`	`end program`	`end select`	`end submodule`
`end subroutine`	`end type`	`end where`	`go to`
`in out`	`select case`	`select type`	

2.4 Source form

The statements of which a source program is composed are written on **lines**. Each line may contain up to 132 characters,[1] and usually contains a single statement. Since leading spaces are not significant, it is possible to start all such statements in the first character position, or in any other position consistent with the user's chosen layout. A statement may thus be written as

```
x = (-y + root_of_discriminant)/(2.0*a)
```

In order to be able to mingle suitable comments with the code to which they refer, Fortran allows any line to carry a trailing comment field, following an exclamation mark (!). An example is

```
x = y/a - b   ! Solve the linear equation
```

Any comment always extends to the end of the source line and may include processor-dependent characters (it is not restricted to the Fortran character set, Section 2.2). Any line whose first non-blank character is an exclamation mark, or which contains only blanks, or which is empty, is purely commentary and is ignored by the compiler. Such comment lines may appear anywhere in a program unit, including ahead of the first statement, and even after the final program unit. A **character context** (those contexts defined in Sections 2.6.4 and 11.2) is allowed to contain !, so the ! does not initiate a comment in this case; in all other cases it does.

Since it is possible that a long statement might not be accommodated in the 132 positions allowed in a single line, up to 255 additional continuation lines are allowed.[2] The so-called

[1] Lines containing characters of non-default kind (Section 2.6.4) are subject to a processor-dependent limit.

[2] Such long statements might be generated automatically.

continuation mark is the ampersand (`&`) character, and this is appended to each line that is followed by a continuation line. Thus, the first statement of this section (considerably spaced out) could be written as

```
x =                                                    &
   (-y + root_of_discriminant)                         &
   /(2.0*a)
```

In this book the ampersands will normally be aligned to improve readability. On a non-comment line, if `&` is the last non-blank character or the last non-blank character ahead of the comment symbol `!`, the statement continues from the character immediately preceding the `&`. Continuation is normally to the first character of the next non-comment line, but if the first non-blank character of the next non-comment line is `&`, continuation is to the character following the `&`. For instance, the above statement may be written

```
x =                                                    &
   &(-y + root_of_discriminant)/(2.0*a)
```

In particular, if a token cannot be contained at the end of a line, the first non-blank character on the next non-comment line must be an `&` followed immediately by the remainder of the token.

Comments are allowed to contain any characters, including `&`, so they cannot be continued because a trailing `&` is taken as part of the comment. However, comment lines may be freely interspersed among continuation lines and do not count towards the limit of 255 lines.

In a character context, continuation must be from a line without a trailing comment and to a line with a leading ampersand. This is because both `!` and `&` are permitted both in character contexts and in comments.

No line is permitted to have `&` as its only non-blank character, or as its only non-blank character ahead of !. Such a line is really a comment and becomes a comment if `&` is removed.

It can be convenient to write several statements on one line. The semicolon (`;`) character may always be used as a **statement separator**, for example:

```
a = 0; b = 0; c = 0
```

but must not appear as the first non-blank character of a line unless it is a continuation line. Adjacent semicolons, possibly separated by blanks, are interpreted as one.

Since commentary always extends to the end of the line, it is not possible to insert commentary between statements on a single line. In principle, it is possible to write several long statements one after the other in a solid block of lines, each 132 characters long and with the appropriate semicolons separating the individual statements. In practice, such code is unreadable, and the use of multiple-statement lines should be reserved for trivial cases such as the one shown in this example.

Any Fortran statement (that is not part of a compound statement) may be labelled, in order to be able to identify it. For some statements a label is mandatory. A statement **label** precedes the statement, and is regarded as a token. The label consists of from one to five digits, one of which must be nonzero. An example of a labelled statement is

```
100 continue
```

Leading zeros are not significant in distinguishing between labels. For example, `10` and `010` are equivalent.

2.5 Concept of type

In Fortran it is possible to define and manipulate various **types** of data. For instance, we may have available the value 10 in a program, and assign that value to an integer **scalar** variable denoted by `i`. Both 10 and `i` are of type integer; 10 is a fixed or **constant** value, whereas `i` is a **variable** which may be assigned other values. Integer expressions, such as `i+10`, are available too.

A **data type** consists of a set of data values, a means of denoting those values, and a set of operations that are allowed on them. For the integer data type the values are $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$ between some limits depending on the kind of integer and computer system being used. Such tokens as these are **literal constants**, and each data type has its own form for expressing them. Named scalar variables, such as `i`, may be established. During the execution of a program the value of `i` may change to any valid value, or may become **undefined**, that is have no predictable value. The operations that may be performed on integers are those of usual arithmetic; we can write `1+10` or `i-3` and obtain the expected results. Named constants may be established too; these have values that do not change during execution of the program.

Properties like those just mentioned are associated with all the data types of Fortran, and will be described in detail in this and the following chapters. The language itself contains five data types whose existence may always be assumed. These are known as the **intrinsic types**, whose literal constants form the subject of the next section. Of each intrinsic type there is a default kind and a processor-dependent number of other kinds. Each kind is associated with a non-negative integer value known as the **kind type parameter**. This is used as a means of identifying and distinguishing the various kinds available.

In addition, it is possible to define other data types based on collections of data of the intrinsic types, and these are known as **derived types**. The ability to define data types of interest to the programmer – matrices, geometrical shapes, lists, interval numbers – is a powerful feature of the language, one which permits a high level of **data abstraction**, that is the ability to define and manipulate **data objects** without being concerned about their actual representation in a computer.

2.6 Literal constants of intrinsic type

The intrinsic data types are divided into two classes. The first class contains three **numeric** types which are used mainly for numerical calculations – integer, real, and complex. The second class contains the two **non-numeric** types which are used for such applications as text processing and program control – character and logical. The numerical types are used in conjunction with the usual operators of arithmetic, such as + and −, which will be described in Chapter 3. Each includes a zero, and the value of a signed zero is the same as that of an unsigned zero.[3] The non-numeric types are used with sets of operators specific to each type; for instance, character data may be concatenated. These too will be described in Chapter 3.

[3] Although the representation of data is processor dependent, for the numeric data types the standard defines model representations and means to inquire about the properties of those models. The details are deferred to Section 9.9.

2.6.1 Integer literal constants

The first type of literal constant is the **integer literal constant**. The default kind is simply a signed or unsigned integer value, for example:

```
1
0
-999
32767
+10
```

The **range** of the default integers is not specified in the language, but on a computer with a word size of n bits, is often from -2^{n-1} to $+2^{n-1}-1$. Thus, on a 32-bit computer the range is often from $-2\,147\,483\,648$ to $+2\,147\,483\,647$. However, the maximum integer size is required to have a range of at least 18 decimal digits, implying, on a binary machine (all modern computers), a 64-bit integer.

To be sure that the range will be adequate on any computer requires the specification of the kind of integer by giving a value for the kind type parameter. This is best done through a named integer constant. For example, if the range $-999\,999$ to $999\,999$ is desired, `k6` may be established as a constant with an appropriate value by the statement, fully explained later,

```
integer, parameter :: k6 = selected_int_kind(6)
```

and used in constants thus:

```
-123456_k6
+1_k6
 2_k6
```

Here, `selected_int_kind(6)` is an intrinsic inquiry function call, and it returns a kind parameter value that yields the range $-999\,999$ to $999\,999$ with the least margin (see Section 9.9.4).

On a given processor, it might be known that the kind value needed is 3. In this case, the first of our constants can be written

```
-123456_3
```

but this form is less portable. If we move the code to another processor, this particular value may be unsupported, or might correspond to a different range.

Many implementations use kind values that indicate the number of bytes of storage occupied by a value, but the standard allows greater flexibility. For example, a processor might have hardware only for four-byte integers, and yet support kind values 1, 2, and 4 with this hardware (to ease portability from processors that have hardware for one-, two-, and four-byte integers). However, the standard makes no statement about kind values or their order, except that the kind value is never negative.

The value of the kind type parameter for a given data type on a given processor can be obtained from the `kind` intrinsic function (Section 9.2):

```
kind(1)         ! for the default value
kind(2_k6)      ! for the example
```

and the decimal exponent range (number of decimal digits supported) of a given entity may be obtained from another function (Section 9.9.2), as in

```
range(2_k6)
```

which in this case would return a value of at least 6.

2.6.2 Real literal constants

The second type of literal constant is the **real literal constant**. The default kind is a floating-point form built of some or all of a signed or unsigned integer part, a decimal point, a fractional part, and a signed or unsigned exponent part. One or both of the integer part and fractional part must be present. The exponent part is either absent or consists of the letter `e` followed by a signed or unsigned integer. One or both of the decimal point and the exponent part must be present. An example is

```
-10.6e-11
```

meaning -10.6×10^{-11}, and other legal forms are

```
1.
-0.1
1e-1
3.141592653
```

The default real literal constants are representations of a subset of the real numbers of mathematics, and the standard specifies neither the allowed range of the exponent nor the number of significant digits represented by the processor. Many processors conform to the IEEE standard for floating-point arithmetic and have values of 10^{-37} to 10^{+37} for the range, and a precision of six decimal digits.

To guarantee obtaining a desired range and precision requires the specification of a kind parameter value. For example,

```
integer, parameter :: long = selected_real_kind(9, 99)
```

ensures that the constants

```
1.7_long
12.3456789e30_long
```

have a precision of at least nine significant decimals and an exponent range of at least 10^{-99} to 10^{+99}. The number of digits specified in the significand has no effect on the kind. In particular, it is permitted to write more digits than the processor can in fact use.

As for integers, many implementations use kind values that indicate the number of bytes of storage occupied by a value, but the standard allows greater flexibility. It specifies only that the kind value is never negative. If the desired kind value is known it may be used directly, as in the case

```
1.7_4
```

but the resulting code is then less portable.

The processor must provide at least one representation with more precision than the default, and this second representation may also be specified as `double precision`. We defer the description of this alternative but outmoded syntax to Appendix A.6.

The `kind` function is valid also for real values:

```
kind(1.0)         ! for the default value
kind(1.0_long)    ! for the example
```

In addition, there are two inquiry functions available which return the actual precision and range, respectively, of a given real entity (see Section 9.9.2). Thus, the value of

```
precision(1.7_long)
```

would be at least 9, and the value of

```
range(1.7_long)
```

would be at least 99.

2.6.3 Complex literal constants

Fortran, as a language intended for scientific and engineering calculations, has the advantage of having as a third literal constant type the **complex literal constant.** This is designated by a pair of literal constants, which are either integer or real, separated by a comma and enclosed in parentheses. Examples are

```
(1., 3.2)
(1, .99e-2)
(1.0, 3.7_8)
```

where the first constant of each pair is the real part of the complex number, and the second constant is the imaginary part. If one of the parts is integer, the kind of the complex constant is that of the other part. If both parts are integer, the kind of the constant is that of the default real type. If both parts are real and of the same kind, this is the kind of the constant. If both parts are real and of different kinds, the kind of the constant is that of one of the parts: the part with the greater decimal precision, or the part chosen by the processor if the decimal precisions are identical.

A default complex constant is one whose kind value is that of default real.

The `kind`, `precision`, and `range` functions are equally valid for complex entities.

Note that if an implementation uses the number of bytes needed to store a real as its kind value, the number of bytes needed to store a complex value of the corresponding kind is twice the kind value. For example, if the default real type has kind 4 and needs four bytes of storage, the default complex type has kind 4 but needs eight bytes of storage.

2.6.4 Character literal constants

The fourth type of literal constant is the **character literal constant.** The default kind consists of a string of characters enclosed in a pair of either apostrophes or quotation marks, for example

```
'Anything goes'
"Nuts & bolts"
```

The characters are not restricted to the Fortran set (Section 2.2). Any graphic character supported by the processor is permitted, but not control characters such as ‘newline’. The apostrophes and quotation marks serve as **delimiters**, and are not part of the value of the constant. The value of the constant

```
'STRING'
```

is `STRING`. Note that in character constants the blank character is significant. For example

```
'a string'
```

is not the same as

```
'astring'
```

A problem arises with the representation of an apostrophe or a quotation mark in a character constant. Delimiter characters of one sort may be embedded in a string delimited by the other, as in the examples

```
'He said "Hello"'
"This contains an ' "
```

Alternatively, a doubled delimiter without any embedded intervening blanks is regarded as a single character of the constant. For example

```
'Isn''t it a nice day'
```

has the value `Isn't it a nice day`.

The number of characters in a string is called its **length**, and may be zero. For instance, `''` and `""` are character constants of length zero.

We mention here the particular rule for the source form concerning character constants that are written on more than one line (needed because constants may include the characters ! and `&`): not only must each line that is continued be without a trailing comment, but each continuation line must begin with a continuation mark. Any blanks following a trailing ampersand or preceding a leading ampersand are not part of the constant, nor are the ampersands themselves part of the constant. Everything else, including blanks, is part of the constant. An example is

```
long_string =                                                  &
     'Were I with her, the night would post too soon;          &
    & But now are minutes added to the hours;                  &
    & To spite me now, each minute seems a moon;               &
    & Yet not for me, shine sun to succour flowers!            &
    &   Pack night, peep day; good day, of night now borrow:   &
    &   Short, night, to-night, and length thyself tomorrow.'
```

On any computer the characters have a property known as their **collating sequence**. One may ask the question whether one character occurs before or after another in the sequence. This question is posed in a natural form such as 'Does C precede M?', and we shall see later how this may be expressed in Fortran terms. Fortran requires the computer's collating sequence to satisfy the following conditions:

- `A` is less than `B` is less than `C` ... is less than `Y` is less than `Z`;
- `a` is less than `b` is less than `c` ... is less than `y` is less than `z`;
- `0` is less than `1` is less than `2` ... is less than `8` is less than `9`;
- blank is less than `A` and `Z` is less than `0`, or blank is less than `0` and `9` is less than `A`;
- blank is less than `a` and `z` is less than `0`, or blank is less than `0` and `9` is less than `a`.

Thus, we see that there is no rule about whether the numerals precede or succeed the letters, nor about the position of any of the special characters or the underscore, apart from the rule that blank precedes both partial sequences. Any given computer system has a complete collating sequence, and most computers nowadays use the collating sequence of the ASCII standard (also known as ISO/IEC 646:1991). However, Fortran is designed to accommodate other sequences, notably EBCDIC, so for portability no program should ever depend on any ordering beyond that stated above. Alternatively, Fortran provides access to the ASCII collating sequence on any computer through intrinsic functions (Section 9.6.1), but this access is not so convenient and is less efficient on some computers.

A processor is required to provide access to the default kind of character constant just described. In addition, it may support other kinds of character constants, in particular those of non-European languages, which may have more characters than can be provided in a single byte. For example, a processor might support Kanji with the kind parameter value 2; in this case, a Kanji character constant may be written

```
2_'国内'
```

or

```
kanji_"標準"
```

where `kanji` is an integer named constant with the value 2. We note that, in this case, the kind type parameter exceptionally precedes the constant.[4]

There is no requirement for a processor to provide more than one kind of character, and the standard does not require any particular relationship between the kind parameter values and the character sets and the number of bytes needed to represent them. In fact, all that is required is that each kind of character set includes a blank character. However, where other kinds are available, the intrinsic function `selected_char_kind`, fully described in Section 9.9.4, can be used to select a specific character set.

As for the other data types, the `kind` function gives the actual value of the kind type parameter, as in

```
kind('ASCII')
```

Non-default characters are permitted in comments.

2.6.5 Logical literal constants

The fifth type of literal constant is the **logical literal constant**. The default kind has one of two values, `.true.` and `.false.` . These logical constants are normally used only to initialize logical variables to their required values, as we shall see in Section 3.6.

The default kind has a kind parameter value which is processor dependent. The actual value is available as `kind(.true.)`. As for the other intrinsic types, the kind parameter may be specified by an integer constant following an underscore, as in

```
.false._1
.true._long
```

[4]This is to make it easier for a compiler to support multiple different character sets occurring within a single source file.

Non-default logical kinds are useful for storing logical arrays compactly; we defer further discussion until Section 7.7.

2.6.6 Binary, octal, and hexadecimal constants

A binary, octal, or hexadecimal ('boz') constant is allowed as a literal constant in limited circumstances to provide the bit sequence of a stored integer or real value. It is known as a **'boz' constant**.

A 'boz' constant may take the form of the letter `b` followed by a binary integer in apostrophes, the letter `o` followed by an octal integer in apostrophes, or the letter `z` followed by a hexadecimal integer in apostrophes. Examples are `b'1011'`, `o'715'`, and `z'a2f'`. The letters `b`, `o`, and `z` may be in upper case, as may the hexadecimal digits `a` to `f`. A pair of quotation marks may be used instead of a pair of apostrophes. In all cases, it is an exact representation of a sequence of bits. The length must not exceed the largest storage size of a real or integer value.

A 'boz' constant is permitted only in a `data` statement (Section 8.5.2) and as an actual argument of a small number of intrinsic procedures (Chapter 9).

A binary, octal, or hexadecimal constant may also appear in an internal or external file as a digit string, without the leading letter and the delimiters (see Section 11.3.2).

2.7 Names

A Fortran program **references** many different entities by name. Such names must consist of between one and 63 alphanumeric characters – letters, underscores, and numerals – of which the first must be a letter. There are no other restrictions on the names; in particular there are no reserved words in Fortran. We thus see that valid names are, for example,

```
a
a_thing
x1
mass
q123
real
time_of_flight
```

and invalid names are

```
1a              ! First character is not alphabetic
a thing         ! Contains a blank
$sign           ! Contains a non-alphanumeric character
```

Within the constraints of the syntax, it is important for program clarity to choose names that have a clear significance – these are known as **mnemonic names**. Examples are `day`, `month`, and `year` for variables to store the calendar date.

The use of names to refer to constants, already met in Section 2.6.1, will be fully described in Section 8.3.

2.8 Scalar variables of intrinsic type

We have seen in the section on literal constants (Section 2.6) that there exist five different intrinsic data types. Each of these types may have variables too. The simplest way by which a variable may be declared to be of a particular type is by specifying its name in a **type declaration statement** such as

```
integer   :: i
real      :: a
complex   :: current
logical   :: pravda
character :: letter
```

Here, all the variables have default kind, and `letter` has default length, which is 1. Explicit requirements may also be specified through **type parameters**, as in the examples

```
integer(kind=4)               :: i
real(kind=long)               :: a
character(len=20, kind=1)     :: english_word
character(len=20, kind=kanji) :: kanji_word
```

Character is the only type to have two parameters, and here the two character variables each have length 20. Where appropriate, just one of the parameters may be specified, leaving the other to take its default value, as in the cases

```
character(kind=kanji) :: kanji_letter
character(len=20)     :: english_word
```

The shorter forms

```
integer(4)           :: i
real(long)           :: a
character(20, 1)     :: english_word
character(20, kanji) :: kanji_word
character(20)        :: english_word
```

are available, but note that

```
character(kanji) :: kanji_letter        ! Beware
```

is not an abbreviation for

```
character(kind=kanji) :: kanji_letter
```

because a single unnamed parameter is taken as the length parameter.

2.9 Derived data types

When programming, it is often useful to be able to manipulate objects that are more sophisticated than those of the intrinsic types. Imagine, for instance, that we wished to specify objects representing persons. Each person in our application is distinguished by a name, an age, and an identification number. Fortran allows us to define a corresponding data type in the following fashion:

```
type person
   character(len=10) :: name
   real              :: age
   integer           :: id
end type person
```

This is the definition of the type and is known as a **derived-type definition**. A scalar object of such a type is called a **structure**. In order to create a structure of that type, we write an appropriate type declaration statement, such as

```
type(person) :: you
```

The scalar variable `you` is then a composite object of type `person` containing three separate **components**, one corresponding to the name, another to the age, and a third to the identification number.

As will be described in Sections 3.9 and 3.10, a variable such as `you` may appear in expressions and assignments involving other variables or constants of the same or different types. In addition, each of the components of the variable may be referenced individually using the **component selector** character percent (`%`). The identification number of `you` would, for instance, be accessed as

```
you%id
```

and this quantity is an integer variable which could appear in an expression such as

```
you%id + 9
```

Similarly, if there were a second object of the same type:

```
type(person) :: me
```

the differences in ages could be established by writing

```
you%age - me%age
```

It will be shown in Section 3.9 how a meaning can be given to an expression such as

```
you - me
```

Just as the intrinsic data types have associated literal constants, so too may literal constants of derived type be specified. Their form is the name of the type followed by the constant values of the components, in order and enclosed in parentheses. Thus, the constant

```
person( 'Smith', 23.5, 2541)
```

may be written assuming the derived type defined at the beginning of this section, and could be assigned to a variable of the same type:

```
you = person( 'Smith', 23.5, 2541)
```

Any such **structure constructor** can appear only after the derived-type definition.

A derived type may have a component that is of a previously defined derived type. This is illustrated in Figure 2.1. A variable of type `triangle` may be declared thus

```
type(triangle) :: t
```

Figure 2.1 A derived type with a component of a previously defined derived type.

```
type point
   real :: x, y
end type point
type triangle
   type(point) :: a, b, c
end type triangle
```

and `t` has components `t%a`, `t%b`, and `t%c` all of type `point`; `t%a` has components `t%a%x` and `t%a%y` of type real.

The real and imaginary parts of a complex variable can be accessed as the pseudo-components `re` and `im` for the real and imaginary parts, respectively. For example, `impedance%re` and `impedance%im` are the real and imaginary parts of the complex variable `impedance`. They are also accessible, more inconveniently, by the intrinsic functions `real` and `aimag` (Section 9.3.1).

2.10 Arrays of intrinsic type

Another compound object supported by Fortran is the **array**. An array consists of a rectangular set of elements, all of the same type and type parameters. There are a number of ways in which arrays may be declared; for the moment we shall consider only the declaration of arrays of fixed sizes. To declare an array named `a` of ten real elements, we add the `dimension` **attribute** to the type declaration statement thus:

```
real, dimension(10) :: a
```

The successive elements of the **whole array** `a` are `a(1)`, `a(2)`, `a(3)`, ..., `a(10)`. The number of elements of an array is called its **size**. Each **array element** is a scalar.

Many problems require a more elaborate declaration than one in which the first element is designated 1, and it is possible in Fortran to declare a lower as well as an upper **bound**:

```
real, dimension(-10:5) :: vector
```

This is a vector of 16 elements, `vector(-10)`, `vector(-9)`, ..., `vector(5)`. We thus see that whereas we always need to specify the upper bound, the lower bound is optional, and by default has the value 1.

An array may extend in more than one dimension, and Fortran allows up to 15 dimensions to be specified.[5] For instance,

```
real, dimension(5,4) :: b
```

declares an array with two dimensions, and

```
real, dimension(-10:5, -20:-1, 0:15, -15:0, 16, 16, 16) :: grid
```

[5] If an array is also a coarray (Chapter 17), the limit applies to the sum of the rank and corank.

declares seven dimensions, the first four with explicit lower bounds. It may be seen that the size of this second array is

$$16 \times 20 \times 16 \times 16 \times 16 \times 16 \times 16 = 335\,544\,320,$$

and that arrays of many dimensions can thus place large demands on the memory of a computer: an array

```
real, dimension(10, 10, 10, 10, 10, 10, 10, 10, 10, 10) :: x
```

of four-byte integers requires 40 GB of memory. The number of dimensions of an array is known as its **rank**. Thus, `grid` has a rank of seven. Scalars are regarded as having rank zero. The number of elements along a dimension of an array is known as the **extent** in that dimension. Thus, `grid` has extents 16, 20, ...

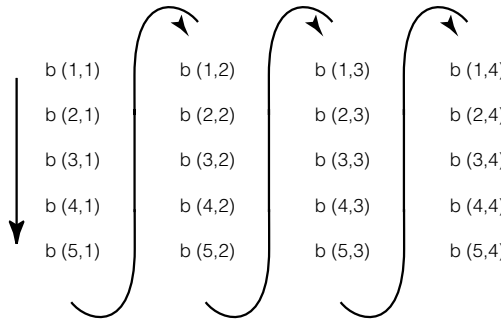
The sequence of extents is known as the **shape**. For example, `grid` has the shape $(16, 20, 16, 16, 16, 16, 16)$.

A derived type may contain an array component. For example, the following type

```
type triplet
   real                  :: u
   real, dimension(3)    :: du
   real, dimension(3,3)  :: d2u
end type triplet
```

might be used to hold the value of a variable in three dimensions and the values of its first and second derivatives. If `t` is of type `triplet`, `t%du` and `t%d2u` are arrays of type real.

Figure 2.2 The ordering of elements in the array `b(5,4)`.



Some statements treat the elements of an array one by one in a special order which we call the **array element order**. It is obtained by counting most rapidly in the early dimensions. Thus, the elements of `grid` in array element order are

```
grid(-10, -20, 0, -15, 1, 1, 1)
grid( -9, -20, 0, -15, 1, 1, 1)
    ⋮
grid(  5,  -1, 15,  0, 16, 16, 16)
```

This is illustrated for an array of two dimensions in Figure 2.2. Most implementations actually store arrays in **contiguous** storage in array element order, but we emphasize that the standard does not require this.

We reference an individual element of an array by specifying, as in the examples above, its **subscript** values. In the examples we used integer constants, but in general each subscript may be formed of a **scalar integer expression**, that is, any arithmetic expression whose value is scalar and of type integer. Each subscript must be within the corresponding ranges defined in the array declaration and the number of subscripts must equal the rank. Examples are

```
a(1)
a(i*j)           ! i and j are of type integer
a(nint(x+3.))    ! x is of type real
t%d2u(i+1,j+2)   ! t is of derived type triplet
```

where `nint` is an intrinsic function to convert a real value to the nearest integer (see Section 9.3.1). In addition subarrays, called **sections**, may be referenced by specifying a range for one or more subscripts. The following are examples of array sections:

```
a(i:j)           ! Rank-one array of size j-i+1
b(k, 1:n)        ! Rank-one array of size n
c(1:i, 1:j, k)   ! Rank-two array with extents i and j
```

We describe array sections in more detail in Section 7.12. An array section is itself an array, but its individual elements must not be accessed through the section designator. Thus, `b(k, 1:n)(l)` cannot be written; it must be expressed as `b(k, l)`.

A further form of subscript is shown in

```
a(ipoint)        ! ipoint is a rank-one integer array
```

where `ipoint` is a rank-one array of indices, pointing to array elements. It may thus be seen that `a(ipoint)`, which identifies as many elements of `a` as `ipoint` has elements, is an example of another **array-valued object**, and `ipoint` is referred to as a **vector subscript**. This will be met in greater detail in Section 7.12.

It is often convenient to be able to define an array constant. In Fortran, a rank-one array may be constructed as a list of elements enclosed between the tokens `(/` and `/)` or the characters `[` and `]`, respectively. A simple example is

```
(/ 1, 2, 3, 5, 10 /)
```

which is an array of rank one and size five. To obtain a series of values, the individual values may be defined by an expression that depends on an integer variable having values in a range, with an optional stride. Thus, the constructor

```
[1, 2, 3, 4, 5]
```

can be written as

```
[ (i, i = 1,5) ]
```

and

```
(/2, 4, 6, 8/)
```

as

```
(/ (i, i = 2,8,2) /)
```

and

```
(/ 1.1, 1.2, 1.3, 1.4, 1.5 /)
```

as

```
(/ (i*0.1, i=11,15) /)
```

An array constant of rank greater than one may be constructed by using the function `reshape` (see Section 9.15.3) to reshape a rank-one array constant.

A full description of array constructors is reserved for Section 7.16.

2.10.1 Declaring entities of differing shapes

In order to declare several entities of the same type but differing shapes, Fortran permits the convenience of using a single statement. Whether or not there is a `dimension` attribute present, arrays may be declared by placing the shape information after the name of the array:

```
integer :: a, b, c(10), d(10), e(8, 7)
```

If the `dimension` attribute is present, it provides a default shape for the entities that are not followed by their own shape information, and is ignored for those that are:

```
integer, dimension(10) :: c, d, e(8, 7)
```

2.10.2 Allocatable objects

Sometimes an object is required to be of a size that is known only after some data have been read or some calculations performed. This can be implemented by the use of pointers (see Section 2.12), but there are significant advantages for memory management and execution speed in using **allocatable objects** when the added functionality of pointers is not needed.

Thus, for this more restricted purpose, an array may be given the `allocatable` attribute by a statement such as

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

Such an array is called **allocatable**. Its rank is specified when it is declared, but the bounds are undefined until an `allocate` statement such as

```
allocate (a(n, 0:n+1))     ! n is of type integer, and is defined
```

has been executed for it.

The allocation status of an allocatable object is either **allocated** or **unallocated**. Its initial status is unallocated and it becomes allocated following successful execution of an `allocate` statement. The object is then available throughout program execution.

When an allocatable object like `a` is no longer needed, it may be deallocated by execution of the statement

```
deallocate (a)
```

following which the object is unallocated.

This important feature is fully described in Chapter 6.

2.11 Character substrings

It is possible to build arrays of characters, just as it is possible to build arrays of any other type:

```
character, dimension(80) :: line
```

declares an array, called `line`, of 80 elements, each one character in length. Each character may be addressed by the usual reference, `line(i)` for example. In this case, however, a more appropriate declaration might be

```
character(len=80) :: line
```

which declares a scalar data object of 80 characters. These may be referenced individually or in groups using a **substring** notation

```
line(i:j)   ! i and j are of type integer
```

which references all the characters from `i` to `j` in `line`. The colon is used to separate the two substring subscripts, which may be any scalar integer expressions. The colon is obligatory in substring references, so that referencing a single character requires `line(i:i)`. There are default values for the substring subscripts. If the lower one is omitted, the value 1 is assumed; if the upper one is omitted, a value corresponding to the character length is assumed. Thus,

`line(:i)` is equivalent to `line(1:i)`
`line(i:)` is equivalent to `line(i:80)`
`line(:)` is equivalent to `line` or `line(1:80)`

If `i` is greater than `j` in `line(i:j)`, the value is a zero-sized string.

We may now combine the length declaration with the array declaration to build arrays of character objects of specified length, as in

```
character(len=80), dimension(60) :: page
```

which might be used to define storage for the characters of a whole page, with 60 elements of an array, each of length 80. To reference the line `j` on a page we may write `page(j)`, and to reference character `i` on that line we could combine the array subscript and character substring notations into

```
page(j)(i:i)
```

A substring of a character constant or of a structure component may also be formed:

```
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'(j:j)
 you%name(1:2)
```

In this section we have used character variables with a declared maximum length. This is adequate for most character manipulation applications, but the limitation may be avoided with allocatable variables, as described in Section 15.4.2.

2.12 Pointers

In everyday language nouns are often used in a way that makes their meaning precise only because of the context. 'The chairman said that ...' will be understood precisely by the reader who knows that the context is the Fortran Committee developing Fortran 90 and that its chairman was then Jeanne Adams.

Similarly, in a computer program it can be very useful to be able to use a name that can be made to refer to different objects during execution of the program. One example is the multiplication of a vector by a sequence of square matrices. We might write code that calculates

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1,2,\ldots,n$$

from the vector $x_j, j = 1,2,\ldots,n$. In order to use this to calculate

$$BCz$$

we might first make x refer to z and A refer to C, thereby using our code to calculate $y = Cz$, then make x refer to y and A refer to B so that our code calculates the result vector we finally want.

An object that can be made to refer to other objects in this way is called a **pointer**, and must be declared with the pointer attribute, for example

```
real, pointer                  :: son
real, pointer, dimension(:)    :: x, y
real, pointer, dimension(:,:) :: a
```

In the case of an array, only the rank (number of dimensions) is declared, and the bounds (and hence shape) are taken from that of the object to which it points. Given such an array pointer declaration, the compiler arranges storage for a descriptor that will later hold the address of the actual object (known as the **target**) and holds, if it is an array, its bounds and strides.

Besides pointing to existing variables, a pointer may be made explicitly to point at nothing:

```
nullify (son, x, y, a)
```

(`nullify` is described in Section 3.14) or may be given fresh storage by an `allocate` statement such as

```
allocate (son, x(10), y(-10:10), a(n, n))
```

In the case of arrays, the lower and upper bounds are specified just as for the `dimension` attribute (Section 2.10), except that any scalar integer expression is permitted. This use of pointers provides a means to access dynamic storage but, as seen in Section 2.10.2 and described later in Chapter 6, allocatable arrays provide a better way to to do this in cases where the 'pointing' property is not essential.

By default, pointers are initially undefined (see also the final paragraph of Section 3.3). This is a very undesirable state since there is no way to test for it. However, it may be avoided by using the declaration:

```
real, pointer :: son => null()
```

(the function `null` is described in Section 9.17) and we recommend that this always be employed. Alternatively, pointers may be defined as soon as they come into scope by execution of a nullify statement or a pointer assignment.

Components of derived types are permitted to have the pointer attribute. This enables a major application of pointers: the construction of linked lists. As a simple example, we might decide to hold a sparse vector as a chain of variables of the type shown in Figure 2.3, which allows us to access the entries one by one; given

```
type(entry), pointer :: chain
```

where `chain` is a scalar of this type and holds a chain that is of length two, its entries are `chain%index` and `chain%next%index`, and `chain%next%next` will have been nullified. Additional entries may be created when necessary by an appropriate allocate statement. We defer the details to Section 3.13.

Figure 2.3 A type for a holding a sparse vector as a chain of variables.

```
type entry
   real                 :: value
   integer              :: index
   type(entry), pointer :: next
end type entry
```

When a pointer is of derived type and a component such as `chain%index` is selected, it is actually a component of the pointer's target that is selected.

A subobject is not a pointer unless it has a final component selector for the name of a pointer component, for example, `chain%next` (whereas `chain%value` and `x(1)` are not).

Pointers will be discussed in detail in later chapters (especially Sections 3.13, 5.7.2, 7.13, 7.14, 8.5.3, 8.5.4, and 9.2).

2.13 Objects and subobjects

We have seen that derived types may have components that are arrays, as in

```
type triplet
   real, dimension(3) :: vertex
end type triplet
```

and arrays may be of derived type as in the example

```
type(triplet), dimension(10) :: t
```

A single structure (for example, `t(2)`) is always regarded as a scalar, but it may have a component (for example, `t(2)%vertex`) that is an array. Derived types may have components of other derived types. The term **ultimate component** is used for a subobject obtained by repeated qualification until there are no further components or we reach an allocatable or pointer component.

An object referenced by an unqualified name (all characters alphanumeric) is called a **named object** and is not part of a bigger object. Its **subobjects** have **designators** that consist

of the name of the **base object** followed by one or more qualifiers (for example, `t(1:7)` and `t(1)%vertex`). Each successive qualifier specifies a part of the base object specified by the name or designator that precedes it.

We note that the term **array** is used for any object that is not scalar, including an array section or an array-valued component of a structure. The term **variable** is used for any named object that is not specified to be a constant and for any part of such an object, including array elements, array sections, structure components, and substrings.

2.14 Summary

In this chapter we have introduced the elements of the Fortran language. The character set has been listed, and the manner in which sequences of characters form literal constants and names explained. In this context we have encountered the five intrinsic data types defined in Fortran, and seen how each data type has corresponding literal constants and named objects. We have seen how derived types may be constructed from the intrinsic types. We have introduced one method by which arrays may be declared, and seen how their elements may be referenced by subscript expressions. The concepts of the array section, character substring, and pointer have been presented, allocatable arrays have been introduced, and some important terms defined. In the following chapter we shall see how these elements may be combined into expressions and statements, Fortran's equivalents of 'phrases' and 'sentences'.

Exercises

1. For each of the following assertions, state whether it is true, false, or not determined, according to the Fortran collating sequences:

`b` is less than `m`
`8` is less than `2`
`*` is greater than `T`
`$` is less than `/`
blank is greater than `A`
blank is less than `6`

2. Which of the Fortran lines in the code

```
   x = y
3  a = b+c ! add
   word = 'string'
   a = 1.0; b = 2.0
   a = 15. ! initialize a; b = 22. ! and b
   song = "Life is just&
      & a bowl of cherries"
   chide = 'Waste not,
      want not!'
0  c(3:4) = 'up"
```

are correctly written according to the requirements of the Fortran source form? Which ones contain commentary? Which lines are initial lines and which are continuation lines?

3. Classify the following literal constants according to the five intrinsic data types of Fortran. Which are not legal literal constants?

```
-43              'word'
4.39              1.9-4
0.0001e+20        'stuff & nonsense'
4 9               (0.,1.)
(1.e3,2)          'I can''t'
'(4.3e9, 6.2)'    .true._1
e5                'shouldn' 't'
1_2                "O.K."
z10                z'10'
```

4. Which of the following names are legal Fortran names?

```
name        name32
quotient    123
a182c3      no-go
stop!       burn_
no_go       long__name
```

5. What are the first, tenth, eleventh, and last elements of the following arrays?

```
real, dimension(11)       :: a
real, dimension(0:11)     :: b
real, dimension(-11:0)    :: c
real, dimension(10,10)    :: d
real, dimension(5,9)      :: e
real, dimension(5,0:1,4) :: f
```

Write an array constructor of eleven integer elements.

6. Given the array declaration

```
character(len=10), dimension(0:5,3) :: c
```

which of the following subobject designators are legal?

```
c(2,3)           c(4:3)(2,1)
c(6,2)           c(5,3)(9:9)
c(0,3)           c(2,1)(4:8)
c(4,3)(:)        c(3,2)(0:9)
c(5)(2:3)        c(5:6)
c(5,3)(9)        c(,)
```

7. Write derived-type definitions appropriate for:

i) a vehicle registration;

ii) a circle;

iii) a book (title, author, and number of pages).

Give an example of a derived type constant for each one.

8. Given the declaration for `t` in Section 2.13, which of the following objects and subobjects are arrays?

```
t                 t(4)%vertex(1)
t(10)             t(5:6)
t(1)%vertex       t(5:5)
```

9. Write specifications for these entities:

a) an integer variable inside the range -10^{20} to 10^{20};

b) a real variable with a minimum of 12 decimal digits of precision and a range of 10^{-100} to 10^{100};

c) a Kanji character variable on a processor that supports Kanji with `kind=2`.

3. Expressions and assignments

3.1 Introduction

We have seen in the previous chapter how we are able to build the 'words' of Fortran – the constants, keywords, and names – from the basic elements of the character set. In this chapter we shall discover how these entities may be further combined into 'phrases' or **expressions**, and how these, in turn, may be combined into 'sentences' or **statements**.

In an expression, we describe a computation that is to be carried out by the computer. The result of the computation may then be assigned to a variable. A sequence of assignments is the way in which we specify, step by step, the series of individual computations to be carried out in order to arrive at the desired result. There are separate sets of rules for expressions and assignments, depending on whether the **operands** in question are numeric, logical, character, or derived in type, and whether they are scalars or arrays. There are also separate rules for pointer assignments. We shall discuss each set of rules in turn, including a description of the relational expressions that produce a result of type logical and are needed in control statements (see next chapter). To simplify the initial discussion, we commence by considering expressions and assignments that are intrinsically defined and involve neither arrays nor entities of derived data types.

An expression in Fortran is formed of operands and operators, combined in a way that follows the rules of Fortran syntax. A simple expression involving a **dyadic** (or **binary**) operator has the form

operand operator operand

an example being

```
x+y
```

and a **unary** or **monadic** operator has the form

operator operand

an example being

```
-y
```

The type and kind of the result are determined by the type and kind of the operands and do not depend on their values. The operands may be constants, variables, or functions (see Chapter 5), and an expression may itself be used as an operand. In this way we can build up more complicated expressions such as

operand operator operand operator operand

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

where consecutive operands are separated by a single operator. Each operand must have a defined value.

The rules of Fortran state that the parts of expressions without parentheses are evaluated successively from left to right for operators of equal precedence, with the exception of `**` (exponentiation, see Section 3.2). If it is necessary to evaluate one part of an expression, or **subexpression,** before another, parentheses may be used to indicate which subexpression should be evaluated first. In

operand operator (*operand operator operand*)

the subexpression in parentheses will be evaluated, and the result used as an operand to the first operator.

If an expression or subexpression has no parentheses, the processor is permitted to evaluate an equivalent expression; that is, an expression that always has the same value apart, possibly, from the effects of numerical round-off. For example, if `a`, `b`, and `c` are real variables, the expression

```
a/b/c
```

might be evaluated as

```
a/(b*c)
```

on a processor that can multiply much faster than it can divide. Usually, such changes are welcome to the programmer since the program runs faster, but when they are not (for instance, because they would lead to more round-off) parentheses should be inserted because the processor is required to respect them.

If two operators immediately follow each other, as in

operand operator operator operand

the only possible interpretation is that the second operator is unary. Thus, there is a general rule that a binary operator must not follow immediately after another operator.

3.2 Scalar numeric expressions

A **numeric expression** is an expression whose operands are one of the three numeric types – integer, real, and complex – and whose operators are

`**`	exponentiation
`* /`	multiplication, division
`+ -`	addition, subtraction

These operators are known as **numeric intrinsic** operators, and are listed here in their order of precedence. In the absence of parentheses, exponentiations will be carried out before multiplications and divisions, and these before additions and subtractions.

We note that the minus sign (-) and the plus sign (+) can be used as unary operators, as in

```
-tax
```

Because it is not permitted in ordinary mathematical notation, a unary minus or plus must not follow immediately after another operator. When this is needed, as for x^{-y}, parentheses must be placed around the operator and its operand:

```
x**(-y)
```

The type and kind type parameter of the result of a unary operation are those of the operand.

The exception to the left-to-right rule noted in Section 3.1 concerns exponentiations. Whereas the expression

```
-a+b+c
```

will be evaluated from left to right as

```
((-a)+b)+c
```

the expression

```
a**b**c
```

will be evaluated as

```
a**(b**c)
```

For integer data, the result of any division will be truncated towards zero, that is to the integer value whose magnitude is equal to or just less than the magnitude of the exact result. Thus, the result of

```
 6/3   is    2
 8/3   is    2
-8/3   is   −2
```

This fact must always be borne in mind whenever integer divisions are written. Similarly, the result of

```
2**3   is   8
```

whereas the result of

```
2**(-3)    is    1/(2**3)
```

which is zero.

The rules of Fortran allow a numeric expression to contain numeric operands of differing types or kind type parameters. This is known as a **mixed-mode expression**. Except when raising a real or complex value to an integer power, the object of the weaker (or simpler) of the two data types will be converted, or **coerced**, into the type of the stronger one. The result will also be that of the stronger type. If, for example, we write

```
a*i
```

when `a` is of type real and `i` is of type integer, then `i` will be converted to a real data type before the multiplication is performed, and the result of the computation will also be of type real. The rules are summarized for each possible combination for the operations `+`, `-`, `*`, and `/` in Table 3.1, and for the operation `**` in Table 3.2. The functions `real` and `cmplx` that they reference are defined in Section 9.3.1. In both tables, I stands for integer, R for real, and C for complex.

If both operands are of type integer, the kind type parameter of the result is that of the operand with the greater decimal exponent range, or is processor dependent if the kinds differ

Table 3.1. Type of result of *a* .op. *b*, where .op. is +, −, *, or /.

Type of *a*	Type of *b*	Value of *a* used	Value of *b* used	Type of result
I	I	*a*	*b*	I
I	R	`real`(*a*, `kind`(*b*))	*b*	R
I	C	`cmplx`(*a*, 0, `kind`(*b*))	*b*	C
R	I	*a*	`real`(*b*, `kind`(*a*))	R
R	R	*a*	*b*	R
R	C	`cmplx`(*a*, 0, `kind`(*b*))	*b*	C
C	I	*a*	`cmplx`(*b*, 0, `kind`(*a*))	C
C	R	*a*	`cmplx`(*b*, 0, `kind`(*a*))	C
C	C	*a*	*b*	C

Table 3.2. Type of result of *a***b*.

Type of *a*	Type of *b*	Value of *a* used	Value of *b* used	Type of result
I	I	*a*	*b*	I
I	R	`real`(*a*, `kind`(*b*))	*b*	R
I	C	`cmplx`(*a*, 0, `kind`(*b*))	*b*	C
R	I	*a*	*b*	R
R	R	*a*	*b*	R
R	C	`cmplx`(*a*, 0, `kind`(*b*))	*b*	C
C	I	*a*	*b*	C
C	R	*a*	`cmplx`(*b*, 0, `kind`(*a*))	C
C	C	*a*	*b*	C

but the decimal exponent ranges are the same. If both operands are of type real or complex, the kind type parameter of the result is that of the operand with the greater decimal precision, or is processor dependent if the kinds differ but the decimal precisions are the same. If one operand is of type integer and the other is real or complex, the type parameter of the result is that of the real or complex operand.

Note that a literal constant in a mixed-mode expression is held to its own precision, which may be less than that of the expression. For example, given a variable `a` of kind `long` (Section 2.6.2), the result of `a/1.7` will be less precise than that of `a/1.7_long`.

In the case of raising a complex value to a complex power, the principal value[1] is taken. Raising a negative real value to a real power is not permitted since the exact result probably has a nonzero imaginary part.

[1]The principal value of a^b is $\exp(b(\log|a| + i\arg a))$, with $-\pi < \arg a \leq \pi$.

3.3 Defined and undefined variables

In the course of the explanations in this and the following chapters, we shall often refer to a variable becoming **defined** or **undefined**. In the previous chapter, we showed how a scalar variable may be called into existence by a statement like

```
real :: speed
```

In this simple case, the variable `speed` has, at the beginning of the execution of the program, no defined value. It is undefined. No attempt must be made to reference its value since it has none. A common way in which it might become defined is for it to be assigned a value:

```
speed = 2.997
```

After the execution of such an **assignment statement** it has a value, and that value may be referenced, for instance in an expression:

```
speed*0.5
```

For a compound object, all of its subobjects that are not pointers must be individually defined before the object as a whole is regarded as defined. Thus, an array is said to be defined only when each of its elements is defined, an object of a derived data type is defined only when each of its non-pointer components is defined, and a character variable is defined only when each of its characters is defined.

A variable that is defined does not necessarily retain its state of definition throughout the execution of a program. As we shall see in Chapter 5, a variable that is local to a single subprogram usually becomes undefined when control is returned from that subprogram. In certain circumstances, it is even possible that a single array element becomes undefined and this causes the array considered as a whole to become undefined; a similar rule holds for entities of derived data type and for character variables.

A means to specify the initial value of a variable is explained in Section 8.5.

In the case of a pointer, the **pointer association** status may be **undefined** or **defined** by being **associated** with a target or being **disassociated**, which means that it is not associated with a target but has a definite status that may be tested by the function `associated` (Section 9.2). Even though a pointer is associated with a target, the target itself may be defined or undefined. Means to specify the initial status of disassociated are provided (see Section 8.5.3).

3.4 Scalar numeric assignment

The general form of a scalar numeric assignment is

variable = *expr*

where *variable* is a scalar numeric variable and *expr* is a scalar numeric expression. If *expr* is not of the same type or kind as *variable*, it will be converted to that type and kind before the assignment is carried out, according to the set of rules given in Table 3.3 (the functions `int`, `real`, and `cmplx` are defined in Section 9.3.1).

We note that if the type of *variable* is integer but *expr* is not, then the assignment will result in a loss of precision unless *expr* happens to have an integral value. Similarly, assigning a real

Table 3.3. Numeric conversion for assignment statement *variable* = *expr*.

Type of *variable*	**Value assigned**
integer	`int`(*expr*, `kind`(*variable*))
real	`real`(*expr*, `kind`(*variable*))
complex	`cmplx`(*expr*, `kind=kind`(*variable*))

expression to a real variable of a kind with less precision will also cause a loss of precision to occur, and the assignment of a complex quantity to a non-complex variable involves the loss of the imaginary part. Thus, the values in `i` and `a` following the assignments

```
i = 7.3                  ! i of type default integer
a = (4.01935, 2.12372)   ! a of type default real
```

are 7 and 4.01935, respectively. Also, if a literal constant is assigned to a variable of greater precision, the result will have the accuracy of the constant. For example, given a variable `a` of kind `long` (Section 2.6.2), the result of

```
a = 1.7
```

will be less precise than that of

```
a = 1.7_long
```

3.5 Scalar relational operators

It is possible in Fortran to test whether the value of one numeric expression bears a certain relation to that of another, and similarly for character expressions. The relational operators are

`<`	less than
`<=`	less than or equal
`==`	equal
`/=`	not equal
`>`	greater than
`>=`	greater than or equal

If either or both of the expressions are complex, only the operators `==` and `/=` are available.

The result of such a comparison is one of the default logical values `.true.` or `.false.`, and we shall see in the next chapter how such tests are important in controlling the execution of a program. Examples of relational expressions (for `i` and `j` of type integer, `a` and `b` of type real, and `char1` of type default character) are

`i < 0`	integer relational expression
`a < b`	real relational expression
`a+b > i-j`	mixed-mode relational expression
`char1 == 'Z'`	character relational expression

In the third expression above, we note that the two components are of different numeric types. In this case, and whenever either or both of the two components consist of numeric expressions, the rules state that the components are to be evaluated separately and converted to the type and kind of their sum before the comparison is made. Thus, a relational expression such as

```
a+b <= i-j
```

will be evaluated by converting the result of `(i-j)` to type real.

For character comparisons, the kinds must be the same and the letters are compared from the left until a difference is found or the strings are found to be identical. The result of the comparison is determined by the positions in the collating sequence (see Section 2.6.4) of the first differing characters. If the lengths differ, the shorter one is regarded as being padded with blanks[2] on the right. Two zero-sized strings are considered to be identical. The effect is as for the order of words in a dictionary. For example, the relation `'ab' < 'ace'` is true.

No other form of mixed-mode relational operator is intrinsically available, though such an operator may be defined (Section 3.9). The numeric operators take precedence over the relational operators.

3.6 Scalar logical expressions and assignments

Logical constants, variables, and functions may appear as operands in logical expressions. The logical operators, in decreasing order of precedence, are:

unary operator:

`.not.` logical negation

binary operators:

`.and.`	logical intersection
`.or.`	logical union
`.eqv.` and `.neqv.`	logical equivalence and non-equivalence

If we assume a logical declaration of the form

```
logical :: i,j,k,l
```

then the following are valid logical expressions:

```
.not.j
j .and. k
i .or. l .and.  .not.j
( .not.k .and. j .neqv. .not.l) .or. i
```

In the first expression we note the use of `.not.` as a unary operator. In the third expression, the rules of precedence imply that the subexpression `l.and..not.j` will be evaluated first, and the result combined with `i`. In the last expression, the two subexpressions

[2]Here and elsewhere, the blank padding character used for a non-default type is processor dependent.

`.not.k.and.j` and `.not.l` will be evaluated and compared for non-equivalence. The result of the comparison, `.true.` or `.false.`, will be combined with `i`.

The kind type parameter of the result is that of the operand for `.not.`, and for the others is that of the operands if they have the same kind or processor dependent otherwise.

We note that the `.or.` operator is an inclusive operator; the `.neqv.` operator provides an exclusive logical or (`a.and..not.b .or. .not.a.and.b`).

The result of any logical expression is the value true or false, and this value may then be assigned to a logical variable such as element 3 of the logical array `flag` in the example

```
flag(3) = ( .not. k .eqv. l) .or. j
```

The kind type parameter values of the variable and expression need not be identical.

A logical variable may be set to a predetermined value by an assignment statement:

```
flag(1) = .true.
flag(2) = .false.
```

In the foregoing examples, all the operands and results were of type logical – no other data type is allowed to participate in an intrinsic logical operation or assignment.

The results of several relational expressions may be combined into a logical expression, and assigned, as in

```
real    :: a, b, x, y
logical :: cond
⋮
cond = a>b .or. x<0.0 .and. y>1.0
```

where we note the precedence of the relational operators over the logical operators. If the value of such a logical expression can be determined without evaluating a subexpression, a processor is permitted not to evaluate the subexpression. An example is

```
i<=10 .and. ary(i)==0    ! for a real array ary(10)
```

when `i` has the value 11. However, the programmer must not rely on such behaviour – an out-of-bounds subscript might be referenced if the processor chooses to evaluate the right-hand subexpression before the left-hand one. We return to this topic in Section 5.10.1.

3.7 Scalar character expressions and assignments

The only intrinsic operator for character expressions is the **concatenation** operator `//`, which has the effect of combining two character operands into a single character result. For example, the result of concatenating the two character constants `AB` and `CD`, written as

```
'AB'//'CD'
```

is the character string `ABCD`. The operands must have the same kind parameter values, but may be character variables, constants, or functions. For instance, if `word1` and `word2` are both of default kind and length 4, and contain the character strings `LOOP` and `HOLE`, respectively, the string `POLE` is the result of

```
word1(4:4)//word2(2:4)
```

The length of the result of a concatenation is the sum of the lengths of the operands. Thus, the length of the result of

```
word1//word2//'S'
```

is 9, which is the length of the string `LOOPHOLES`.

The result of a character expression may be assigned to a character variable of the same kind. Assuming the declarations

```
character(len=4) :: char1, char2
character(len=8) :: result
```

we may write

```
char1 = 'any '
char2 = 'book'
result = char1//char2
```

In this case, `result` will now contain the string `any book`. We note in these examples that the lengths of the left- and right-hand sides of the three assignments are in each case equal. If, however, the length of the result of the right-hand side is shorter than the length of the left-hand side, then the result is placed in the left-most part of the left-hand side and the rest is filled with blank characters. Thus, in

```
character(len=5) :: fill
fill(1:4) = 'AB'
```

`fill(1:4)` will have the value `AB`*bb* (where *b* stands for a blank character). The value of `fill(5:5)` remains undefined, that is, it contains no specific value and should not be used in an expression. As a consequence, `fill` is also undefined. On the other hand, when the left-hand side is shorter than the result of the right-hand side, the right-hand end of the result is truncated. The result of

```
character(len=5) :: trunc8
trunc8 = 'TRUNCATE'
```

is to place in `trunc8` the character string `TRUNC`. If a left-hand side is of zero length, no assignment takes place.

The left- and right-hand sides of an assignment may overlap. In such a case, it is always the old values that are used in the right-hand side expression. For example, the assignment

```
result(3:5) = result(1:3)
```

is valid and if `result` began with the value `ABCDEFGH`, it would be left with the value `ABABCFGH`.

Other means of manipulating characters and strings of characters, via intrinsic functions, are described in Sections 9.6 and 9.7.

3.7.1 ASCII character set

If the default character set for a processor is not ASCII, but ASCII is supported on that processor, intrinsic assignment is defined between them to convert characters appropriately. For example, on an EBCDIC machine, in

```
integer, parameter :: ascii = selected_char_kind('ASCII')
character          :: ce
character(ascii)   :: ca
ce = ascii_'X'
ca = 'X'
```

the first assignment statement will convert the ASCII upper-case X to an EBCDIC upper-case X, and the second assignment statement will do the reverse.

3.7.2 ISO 10646 character set

ISO/IEC 10646 UCS-4 is a four-byte character set designed to be able to represent every character in every language in the world, including all special characters in use in other coded character sets. It is a strict superset of seven-bit ASCII; that is, its first 128 characters are the same as those of ASCII.

Assignment of default characters or ASCII characters to ISO 10646 is allowed, and the characters are converted appropriately. Assignment of ISO 10646 characters to default or ASCII characters is also allowed; however, if any ISO 10646 character is not representable in the destination character set, the result is processor dependent (information will be lost).

For example, in

```
integer, parameter :: ascii = selected_char_kind('ASCII')
integer, parameter :: iso10646 = selected_char_kind('ISO_10646')
character(ascii) :: x = ascii_'X'
character(iso10646) :: y
y = x
```

the ISO 10646 character variable `y` will be set to the correct value for the upper-case letter X.

3.8 Structure constructors

A structure may be constructed from expressions for its components, just as a constant structure may be constructed from constants (Section 2.9). The general form of a **structure constructor** is

derived-type-spec (*[component-spec-list]*)

where *derived-type-spec* is the name of a derived type[3] and each *component-spec* is

[keyword =] expr

or

[keyword =] target

where *keyword* is the name of a component of the type, *expr* is a value for a non-pointer component, and *target* is a target for a pointer component.

In a simple case, the *component-spec*s are without *keyword*s and provide values for the components in the order they appear in the type declaration. For example, given the type

[3] A more general form will be discussed in Section 13.2.

```
type char10
   integer           :: length
   character(len=10) :: value
end type char10
```

and the variables

```
character(len=4) :: char1, char2
```

the following is a value of type `char10`:

```
char10(8, char1//char2)
```

If *keyword*s are used they must appear after any *component-spec*s without keywords but may be in any order. No component may appear more than once. A component may be absent only if it has default initialization (Section 8.5.5).

Non-pointer components are treated as if the intrinsic assignment statement

derived-type-spec`%`*keyword* `=` *expr*

had been executed, and pointer components are treated as if the pointer assignment statement

derived-type-spec`%`*keyword* `=>` *target*

had been executed. In both cases, the rules for intrinsic assignment and pointer assignment apply.[4]

3.9 Scalar defined operators

No operators for derived types are automatically available. If a programmer defines a derived type and wishes operators to be available, he or she must define the operators, too. For a binary operator this is done by writing a function, with two intent `in` arguments, that specifies how the result depends on the operands, and an **interface block** that associates the function with the operator token (functions, intent, and interface blocks will be explained fully in Chapter 5). For example, given the type

```
type interval
   real :: lower, upper
end type interval
```

that represents intervals of numbers between a lower and an upper bound, we may define addition by a module (Section 5.5) containing the procedure

```
function add_interval(a,b)
   type(interval)             :: add_interval
   type(interval), intent(in) :: a, b
   add_interval%lower = a%lower + b%lower ! Production code would
   add_interval%upper = a%upper + b%upper ! allow for roundoff.
end function add_interval
```

and the interface block (Section 5.18)

[4] In particular, *target* must not be a constant.

```
interface operator(+)
   module procedure add_interval
end interface
```

This function would be invoked in an expression such as

```
y + z
```

to perform this programmer-defined add operation for scalar variables `y` and `z` of type `interval`. A unary operator is defined by an interface block and a function with one intent `in` argument.

The operator token may be any of the tokens used for the intrinsic operators or may be a sequence of up to 63 letters enclosed in decimal points other than `.true.` or `.false.`. An example is

```
.sum.
```

In this case, the header line of the interface block would be written as

```
interface operator(.sum.)
```

and the expression as

```
y.sum.z
```

If an intrinsic token is used, the number of arguments must be the same as for the intrinsic operation, the precedence of the operation is as for the intrinsic operation, and a unary minus or plus must not follow immediately after another operator. Otherwise, it is of highest precedence for defined unary operators and lowest precedence for defined binary operators. The complete set of precedences is given in Table 3.4. Where another precedence is required within an expression, parentheses must be used.

Table 3.4. Relative precedence of operators (in decreasing order).

Type of operation when intrinsic	**Operator**
—	monadic (unary) defined operator
Numeric	`**`
Numeric	`*` or `/`
Numeric	monadic `+` or `-`
Numeric	dyadic `+` or `-`
Character	`//`
Relational	`==` `/=` `<` `<=` `>` `>=`
Logical	`.not.`
Logical	`.and.`
Logical	`.or.`
Logical	`.eqv.` or `.neqv.`
—	dyadic (binary) defined operator

Retaining the intrinsic precedences is helpful both to the readability of expressions and to the efficiency with which a compiler can interpret them. For example, if + is used for set union and * for set intersection, we can interpret the expression

```
i*j + k
```

for sets `i`, `j`, and `k` without difficulty.

If either of the intrinsic tokens == and `.eq.` is used, the definition applies to both tokens so that they are always equivalent. The same is true for the other equivalent pairs of relational operators.

Note that a defined unary operator not using an intrinsic token may follow immediately after another operator, as in

```
y .sum. .inverse. x
```

Operators may be defined for any types of operands, except where there is an intrinsic operation for the operator and types. For example, we might wish to be able to add an interval number to an ordinary real, which can be done with the extra procedure

```
function add_interval_real(a,b)
   type(interval)             :: add_interval_real
   type(interval), intent(in) :: a
   real, intent(in)           :: b
   add_interval_real%lower = a%lower + b ! Production code would
   add_interval_real%upper = a%upper + b ! allow for roundoff.
end function add_interval_real
```

changing the interface block to

```
interface operator(+)
   module procedure add_interval, add_interval_real
end interface
```

The result of such a **defined operation** may have any type. The type of the result, as well as its value, must be specified by the function.

Note that an operation that is defined intrinsically cannot be redefined; thus in

```
real :: a, b, c
⋮
c = a + b
```

the meaning of the operation is always unambiguous.

3.10 Scalar defined assignments

Assignment of an expression of derived type to a variable of the same type is automatically available and takes place component by component. For example, if `a` is of the type `interval` defined at the start of Section 3.9, we may write

```
a = interval(0.0, 1.0)
```

(structure constructors were met in Section 3.8).

In other circumstances, however, we might wish to define a different action for an assignment involving an object of derived type, and indeed this is possible. An assignment may be redefined or another assignment may be defined by a subroutine with two arguments, the first having intent `out` or intent `inout` and corresponding to the variable, and the second having intent `in` and corresponding to the expression (subroutines will also be dealt with fully in Chapter 5). In the case of an assignment involving an object of derived type and an object of a different type, such a definition must be provided. For example, assignment of reals to intervals and vice versa might be defined by a module containing the subroutines

```
subroutine real_from_interval(a,b)
   real, intent(out)          :: a
   type(interval), intent(in) :: b
   a = (b%lower + b%upper)/2
end subroutine real_from_interval
```

and

```
subroutine interval_from_real(a,b)
   type(interval), intent(out) :: a
   real, intent(in)            :: b
   a%lower = b
   a%upper = b
end subroutine interval_from_real
```

and the interface block

```
interface assignment(=)
   module procedure real_from_interval, interval_from_real
end interface
```

Given this, we may write

```
type(interval) :: a
a = 0.0
```

A **defined assignment** must not redefine the meaning of an **intrinsic assignment** for intrinsic types, that is, an assignment between two objects of numeric type, of logical type, or of character type with the same kind parameter, but may redefine the meaning of an intrinsic assignment for two objects of the same derived type. For instance, for an assignment between two variables of the type `char10` (Section 3.9) that copies only the relevant part of the character component, we might write

```
subroutine assign_string (left, right)
   type(char10), intent(out) :: left
   type(char10), intent(in)  :: right
   left%length = right%length
   left%value(1:left%length) = right%value(1:right%length)
end subroutine assign_string
```

Intrinsic assignment for a derived-type object always involves intrinsic assignment for all its non-pointer components, even if a component is of a derived type for which assignment has been redefined.

3.11 Array expressions

So far in this chapter we have assumed that all the entities in an expression are scalar. However, any of the unary intrinsic operations may also be applied to an array to produce another array of the same shape (identical rank and extents, see Section 2.10) and having each element value equal to that of the operation applied to the corresponding element of the operand. Similarly, binary intrinsic operations may be applied to a pair of arrays of the same shape to produce an array of that shape, with each element value equal to that of the operation applied to corresponding elements of the operands. One of the operands of a binary operation may be a scalar, in which case the result is as if the scalar had been broadcast to an array of the same shape as the array operand. Given the array declarations

```
real, dimension(10,20) :: a,b
real, dimension(5)     :: v
```

the following are examples of array expressions:

```
a/b          ! Array of shape [10,20], with elements a(i,j)/b(i,j)
v+1.         ! Array of shape [5], with elements v(i)+1.0
5/v+a(1:5,5) ! Array of shape [5], with elements 5/v(i)+a(i,5)
a == b       ! Logical array of shape [10,20], with element (i,j)
             ! .true. if a(i,j) == b(i,j), and .false. otherwise
```

Two arrays of the same shape are said to be **conformable**, and a scalar is conformable with any array.

Note that the correspondence is by position in the extent and not by subscript value. For example,

```
a(2:9,5:10) + b(1:8,15:20)
```

has element values

`a(i+1,j+4) + b(i,j+14)` where `i` $= 1, 2, \ldots, 8,$ `j` $= 1, 2, \ldots, 6$

This may be represented pictorially as in Figure 3.1.

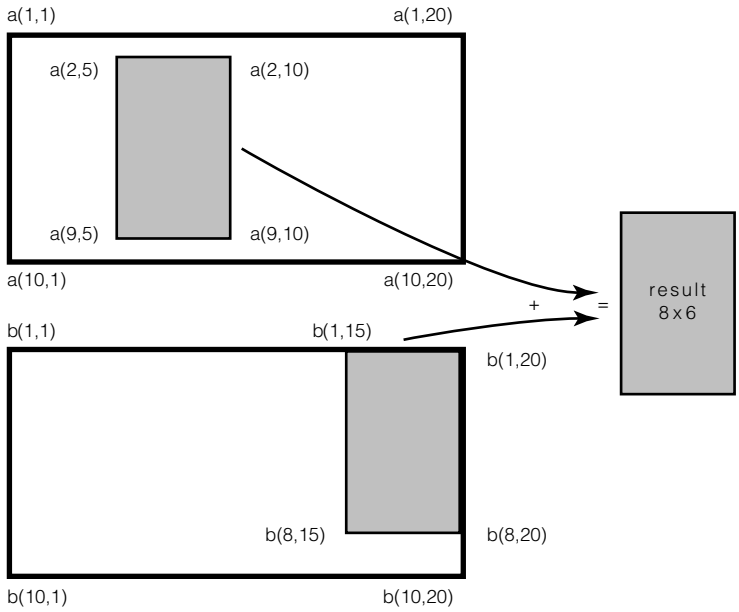
The order in which the scalar operations in any array expression are executed is not specified in the standard, thus enabling a compiler to arrange efficient execution on a vector or parallel computer.

Any scalar intrinsic operator may be applied in this way to arrays and array–scalar pairs. For derived operators, the programmer may define an elemental procedure with these properties (see Section 7.9). He or she may also define operators directly for certain ranks or pairs of ranks. For example, the type

```
type matrix
   real :: element
end type matrix
```

might be defined to have scalar operations that are identical to the operations for reals, but for arrays of ranks one and two the operator `*` defined to mean matrix multiplication. The type `matrix` would therefore be suitable for matrix arithmetic, whereas reals are not suitable because multiplication for real arrays is done element by element. This is further discussed in Section 7.5.

Figure 3.1 The sum of two array sections.



3.12 Array assignment

By intrinsic assignment, an array expression may be assigned to an array variable of the same shape,[5] which is interpreted as if each element of the expression were assigned to the corresponding element of the variable. For example, with the declarations of the beginning of the last section, the assignment

```
a = a + 1.0
```

replaces `a(i,j)` by `a(i,j)`+1.0 for `i` = 1, 2, ..., 10 and `j` = 1, 2, ..., 20. Note that, as for expressions, the element correspondence is by position within the extent rather than by subscript value. This is illustrated by the example

```
a(1,11:15) = v      ! a(1,j+10) is assigned from
                    ! v(j), j = 1, 2, ..., 5
```

A scalar expression may be assigned to an array, in which case the scalar value is broadcast to all the array elements.

If the expression includes a reference to the array variable or to a part of it, the expression is interpreted as being fully evaluated before the assignment commences. For example, the statement

```
v(2:5) = v(1:4)
```

[5]For allocatable arrays the shapes may differ, see Section 6.7.

results in each element `v(i)` for `i` = 2, 3, 4, 5 having the value that `v(i-1)` had prior to the commencement of the assignment. This rule exactly parallels the rule for substrings that was explained in Section 3.7. The order in which the array elements are assigned is not specified by the standard, to allow optimizations.

Sets of numeric and mathematical intrinsic functions, whose results may be used as operands in scalar or array expressions and in assignments, are described in Sections 9.3 and 9.4.

If a defined assignment (Section 3.10) is defined by an elemental subroutine (Section 7.9), it may be used to assign a scalar value to an array or an array value to an array of the same shape. A separate subroutine may be provided for any particular combination of ranks and will override the elemental assignment. If there is no elemental defined assignment, intrinsic assignment is still available for those combinations of ranks for which there is no corresponding defined assignment.

A form of array assignment under a mask is described in Section 7.6 and assignment expressed with the help of indices in Appendix B.3.1.

3.13 Pointers in expressions and assignments

A pointer may appear as a variable in the expressions and assignments that we have considered so far in this chapter, provided it has a valid association with a target. The target is accessed without any need for an explicit dereferencing symbol. In particular, if both sides of an assignment statement are pointers, data are copied from one target to the other target.

Sometimes the need arises for another sort of assignment. We may want the left-hand pointer to point to another target, rather than that its current target acquire fresh data. That is, we want the descriptor to be altered. This is called **pointer assignment** and takes place in a pointer assignment statement:

pointer `=>` *target*

where *pointer* is the name of a pointer or the designator of a structure component that is a pointer, and *target* is usually a variable but may also be a reference to a pointer-valued function (see Section 5.10). For example, the statements

```
x => z
a => c
```

have variables as targets and are needed for the first matrix multiplication of Section 2.12, in order to make `x` refer to `z` and `a` to refer to `c`. The statement

```
x => null()
```

(the function `null` is described in Section 9.17) nullifies `x`. Pointer assignment also takes place for a pointer component of a structure when the structure appears on the left-hand side of an ordinary assignment. For example, suppose we have used the type `entry` of Figure 2.3 of Section 2.12 to construct a chain of entries and wish to add a fresh entry at the front. If `first` points to the first entry and `current` is a scalar pointer of type `entry`, the statements

```
allocate (current)
current = entry(new_value, new_index, first)
first => current
```

allocate a new entry and link it into the top of the chain. The assignment statement has the effect

```
current%next => first
```

and establishes the link. The pointer assignment statement gives `first` the new entry as its target without altering the old first entry. The ordinary assignment

```
first = current
```

would be incorrect because the target would be copied, destroying the old first entry, corresponding to the component assignments

```
first%value = current%value     ! Components of the
first%index = current%index     ! old first are lost.
first%next => current%next      !
```

In the case where the chain began with length two and consisted of

```
first :      (1.0, 10, associated)
first%next : (2.0, 15, null)
```

following the execution of the first set of statements it would have length three and consist of

```
first :           (4.0, 16, associated)
first%next :      (1.0, 10, associated)
first%next%next : (2.0, 15, null)
```

If the *target* in a pointer assignment statement is a variable that is not itself a pointer or a subobject of a pointer target, it must have the `target` attribute. For example, the statement

```
real, dimension(10), target :: y
```

declares `y` to have the `target` attribute. Any non-pointer subobject of an object with the `target` attribute also has the `target` attribute. The `target` attribute is required for the purpose of code optimization by the compiler. It is very helpful to the compiler to know that a variable that is not a pointer or a target may not be accessed by a pointer target.

The target in a pointer assignment statement may be a subobject of a pointer target. For example, given the declaration

```
character(len=80), dimension(:), pointer :: page
```

and an appropriate association, the following are all permitted targets:

```
page, page(10), page(2:4), page(2)(3:15)
```

Note that it is sufficient for the pointer to be at any level of component selection. For example, given the declaration

```
type(entry) :: node
```

which has a pointer component `next` (see Section 2.12) and an appropriate association, `node%next%value` is a permitted target.

If the *target* in a pointer assignment statement is itself a pointer target, then a straightforward copy of the descriptor takes place. If the pointer association status is undefined or disassociated, this state is copied.

If the *target* is a pointer or a subobject of a pointer target, the new association is with that pointer's target and is not affected by any subsequent changes to its pointer association status. This is illustrated by the following example. The sequence

```
b => c    ! c has the target attribute
a => b
nullify (b)
```

will leave `a` still pointing to `c`.

The type, type parameters, and rank of the *pointer* and *target* in a pointer assignment statement must each be the same. If the *pointer* is an array, it takes its shape and bounds from the *target*. The bounds are as would be returned by the functions `lbound` and `ubound` (Section 9.14.2) for the target, which means that an array section or array expression is usually taken to have the value 1 for a lower bound and the extent for the corresponding upper bound (but we shall see later how a lower bound may be specified, see Section 7.15).

Fortran is unusual in not requiring a special character for a reference to a pointer target, but requiring one for distinguishing pointer assignment from ordinary assignment. The reason for this choice was the expectation that most engineering and scientific programs will refer to target data far more often than they change targets.

3.14 The nullify statement

A pointer may be explicitly disassociated from its target by executing a `nullify` statement. Its general form is

`nullify(`*pointer-object-list*`)`

There must be no dependencies among the objects, in order to allow the processor to nullify the objects one by one in any order. The statement is also useful for giving the disassociated status to an undefined pointer. An advantage of nullifying pointers rather than leaving them undefined is that they may then be tested by the intrinsic function `associated` (Section 9.2). For example, the end of the chain of Section 3.13 will be flagged as a disassociated pointer if the statement

```
nullify(first)
```

is executed initially to create a zero-length chain. Because there are often other ways to access a target (for example, through another pointer), the `nullify` statement does not deallocate the targets. If deallocation is also required, a `deallocate` statement (Section 6.6) should be executed instead.

3.15 Summary

In this chapter we have seen how scalar and array expressions of numeric, logical, character, and derived types may be formed, and how the corresponding assignments of the results may be made. The relational expressions and the use of pointers have also been presented. We now have the information required to write short sections of code forming a sequence of statements to be performed one after the other. In the following chapter we shall see how more complicated sequences, involving branching and iteration, may be built up.

Exercises

1. If all the variables are numeric scalars, which of the following are valid numeric expressions?

```
a+b              -c
a+-c             d+(-f)
(a+c)**(p+q)     (a+c)(p+q)
-(x+y)**i        4.((a-d)-(a+4.*x)+1)
```

2. In the following expressions, add the parentheses which correspond to Fortran's rules of precedence (assuming `a`, `c`–`f` are real scalars, `i`–`n` are logical scalars, and `b` is a logical array); for example, `a+d**2/c` becomes `a+((d**2)/c)`.

```
c+4.*f
4.*g-a+d/2.
a**e**c**d
a*e-c**d/a+e
i .and. j .or. k
.not. l .or. .not. i .and. m .neqv. n
b(3).and.b(1).or.b(6).or..not.b(2)
```

3. What are the results of the following expressions?

```
3+4/2        6/4/2
3.*4**2      3.**3/2
-1.**2       (-1.)**3
```

4. A scalar character variable `r` has length eight. What are the contents of `r` after each of the following assignments?

```
r = 'ABCDEFGH'
r = 'ABCD'//'01234'
r(:7) = 'ABCDEFGH'
r(:6) = 'ABCD'
```

5. Which of the following logical expressions are valid if `b` is a logical array?

```
.not.b(1).and.b(2)       .or.b(1)
b(1).or..not.b(4)        b(2)(.and.b(3).or.b(4))
```

6. If all the variables are real scalars, which of the following relational expressions are valid?

```
d <= c             p < t > 0
x-1 /= y             x+y < 3 .or. > 4.
d < c.and.3.0       q == r .and. s>t
```

7. Write expressions to compute:

a) the perimeter of a square of side l;

b) the area of a triangle of base b and height h;

c) the volume of a sphere of radius r.

8. An item costs n cents. Write a declaration statement for suitable variables and assignment statements which compute the change to be given from a $1 bill for any value of n from 1 to 99, using coins of denomination 1, 5, 10, and 25 cents.

9. Given the type declaration for `interval` in Section 3.9, the definitions of `+` given in Section 3.9, the definitions of assignment given in Section 3.10, and the declarations

```
type(interval) :: a,b,c,d
real           :: r
```

which of the following statements are valid?

```
a = b + c
c = b + 1.0
d = b + 1
r = b + c
a = r + 2
```

10. Given the type declarations

```
real, dimension(5,6) :: a, b
real, dimension(5)   :: c
```

which of the following statements are valid?

```
a = b                c = a(:,2) + b(5,:5)
a = c+1.0            c = a(2,:) + b(:,5)
a(:,3) = c           b(2:,3) = c + b(:5,3)
```

4. Control constructs

4.1 Introduction

We have learnt in the previous chapter how assignment statements may be written, and how these may be ordered one after the other to form a sequence of code which is executed step by step. In most computations, however, this simple sequence of statements is by itself inadequate for the formulation of the problem. For instance, we may wish to follow one of two possible paths through a section of code, depending on whether a calculated value is positive or negative. We may wish to sum 1000 elements of an array, and to do this by writing 1000 additions and assignments is clearly tedious; the ability to iterate over a single addition is required instead. We may wish to pass control from one part of a program to another, or even stop processing altogether.

For all these purposes, we have available in Fortran various facilities to enable the logical flow through the program statements to be controlled. The most important form is that of a **block construct**, which begins with an initial keyword statement, may have intermediate keyword statements, and ends with a matching terminal statement; it may be entered only at the initial statement. Each sequence of statements between keyword statements is called a **block**. A block may be empty, though such cases are rare.

Block constructs may be **nested**, that is a block may contain another block construct. In such a case, the block must contain the whole of the inner construct. Execution of a block always begins with its first statement.

4.2 The if construct and statement

The `if` construct contains one or more sequences of statements (blocks), at most one of which is chosen for execution. The general form is shown in Figure 4.1. Here and throughout the book we use square brackets to indicate optional items, followed by dots if there may be any number (including zero) of such items. There can be any number (including zero) of `else if` statements, and zero or one `else` statements. Naming is optional, but an `else` or `else if` statement may be named only if the corresponding `if` and `end if` statements are named, and must be given the same name. The name may be any valid and distinct Fortran name (see Section 5.15 for a discussion on the scope of names).

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

Figure 4.1 The `if` construct.

```
[ name: ] if (scalar-logical-expr) then
              block
          [ else if (scalar-logical-expr) then [ name ]
              block ] ...
          [ else [ name ]
              block ]
          end if [ name ]
```

An example of the `if` construct in its simplest form is

```
swap: if (x < y) then
         temp = x
         x = y
         y = temp
      end if swap
```

The block of three statements is executed if the condition is true; otherwise execution continues from the statement following the `end if` statement. Note that the block inside the `if` construct is indented. This is not obligatory, but makes the logic easier to understand, especially in nested `if` constructs as we shall see at the end of this section.

The next simplest form has an `else` block, but no `else if` blocks. Now there is an alternative block for the case where the condition is false. An example is

```
if (x < y) then
   x = -x
else
   y = -y
end if
```

in which the sign of `x` is changed if `x` is less than `y`, and the sign of `y` is changed if `x` is greater than or equal to `y`.

The most general type of `if` construct uses the `else if` statement to make a succession of tests, each of which has its associated block of statements. The tests are made one after the other until one is fulfilled, and the associated statements of the relevant `if` or `else if` block are executed. Control then passes to the end of the `if` construct. If no test is fulfilled, no block is executed, unless there is a final 'catch-all' `else` clause.

There is a useful shorthand form for the simplest case of all. An if construct of the form

```
if (scalar-logical-expr) then
   action-stmt
end if
```

may be written

```
if (scalar-logical-expr) action-stmt
```

Examples are

```
if (x-y > 0.0) x = 0.0
if (cond .or. p<q .and. r<=1.0) s(i,j) = t(j,i)
```

It is permitted to nest `if` constructs to an arbitrary depth, as shown to two levels in Figure 4.2, in which we see the necessity to indent the code in order to be able to understand the logic easily. For even deeper nesting, naming is to be recommended. The constructs must be properly nested; that is, each construct must be wholly contained in a block of the next outer construct.

Figure 4.2 A nested `if` construct.

```
if (i < 0) then
   if (j < 0) then
      x = 0.0
      y = 0.0
   else
      z = 0.0
   end if
else if (k < 0) then
   z = 1.0
else
   x = 1.0
   y = 1.0
end if
```

4.3 The case construct

Fortran provides another means of selecting one of several options, rather similar to that of the `if` construct. The principal differences between the two constructs are that, for the `case` construct, only **one** expression is evaluated for testing, and the evaluated expression may belong to no more than one of a series of predefined sets of values. The form of the `case` construct is shown by:

[name: *]* `select case (`*expr*`)`
[`case` *selector [name]*
block] . . .
`end select` *[name]*

As for the `if` construct, the leading and trailing statements must either both be unnamed or both bear the same name; a `case` statement within it may be named only if the leading statement is named and bears the same name. The expression *expr* must be scalar and of type character, logical, or integer, and the specified values in each *selector* must be of this type. In the character case, the lengths are permitted to differ, but not the kinds. In the logical and integer cases, the kinds may differ. The simplest form of *selector* is a scalar constant expression[1] in parentheses, such as in the statement

[1] A constant expression is a restricted form of expression that can be verified to be constant (the restrictions being chosen for ease of implementation). The details are tedious and are deferred to Section 8.4. In this section, all examples employ the simplest form of constant expression: the literal constant.

```
case (1)
```

For character or integer *expr*, a range may be specified by a lower and an upper scalar constant expression separated by a colon:

```
case (low:high)
```

Either *low* or *high*, but not both, may be absent; this is equivalent to specifying that the case is selected whenever *expr* evaluates to a value that is less than or equal to *high*, or greater than or equal to *low*, respectively. An example is shown in Figure 4.3.

Figure 4.3 A `case` construct.

```
select case (number)       ! number is of type integer
case (:-1)                 ! all values below 0
   n_sign = -1
case (0)                   ! only 0
   n_sign = 0
case (1:)                  ! all values above 0
   n_sign = 1
end select
```

The general form of *selector* is a list of non-overlapping values and ranges, all of the same type as *expr*, enclosed in parentheses, such as

```
case (1, 2, 7, 10:17, 23)
```

The form

```
case default
```

is equivalent to a list of all the possible values of *expr* that are not included in the other selectors of the construct. Though we recommend that the values be in order, as in this example, this is not required. Overlapping values are not permitted within one *selector*, nor between different ones in the same construct.

There can only be a single `case default` *selector* in a given `case` construct, as shown in Figure 4.4. The `case default` clause does not necessarily have to be the last clause of the `case` construct.

Figure 4.4 A `case` construct with a `case default` selector.

```
select case (ch)         ! ch of type character
case ('c', 'd', 'r':)
   ch_type = .true.
case ('i':'n')
   int_type = .true.
case default
   real_type = .true.
end select
```

Since the values of the selectors are not permitted to overlap, at most one selector may be satisfied; if none is satisfied, control passes to the next executable statement following the `end select` statement.

Like the `if` construct, `case` constructs may be nested inside one another.

4.4 The do construct

Many problems in mathematics require the ability to iterate. If we wish to sum the elements of an array `a` of length 10, we could write

```
sum = a(1)
sum = sum+a(2)
⋮
sum = sum+a(10)
```

which is clearly laborious. Fortran provides a facility known as the `do` construct which allows us to reduce these ten lines of code to

```
sum = 0.0
do  i = 1,10 ! i is of type integer
   sum = sum+a(i)
end do
```

In this fragment of code we first set `sum` to zero, and then require that the statement between the `do` statement and the `end do` statement shall be executed ten times. For each iteration there is an associated value of an index, kept in `i`, which assumes the value 1 for the first iteration through the loop, 2 for the second, and so on up to 10. The variable `i` is a normal integer variable, but is subject to the rule that it must not be explicitly modified within the `do` construct.

The `do` statement has more general forms. If we wished to sum the fourth to ninth elements we would write

```
do  i = 4, 9
```

thereby specifying the required first and last values of `i`. If, alternatively, we wished to sum all the odd elements, we would write

```
do  i = 1, 9, 2
```

where the third of the three **loop parameters**, namely the 2, specifies that `i` is to be incremented in steps of 2, rather than by the default value of 1, which is assumed if no third parameter is given. In fact, we can go further still, as the parameters need not be constants at all, but integer expressions, as in

```
    do  i = j+4, m, -k(j)**2
```

in which the first value of `i` is `j+4`, and subsequent values are decremented by `k(j)**2` until the value of `m` is reached. Thus, `do` indices may run 'backwards' as well as 'forwards'. If any of the three parameters is a variable or is an expression that involves a variable, the value of the variable may be modified within the loop without affecting the number of iterations, as the **initial** values of the parameters are used for the control of the loop.

The general form of this type of bounded `do` construct control clause is

[name `:` *]* `do` *[,] variable* = *expr*$_1$, *expr*$_2$ *[, expr*$_3$ *]*
block
`end do` *[name]*

where *variable* is a named scalar integer variable, *expr*$_1$, *expr*$_2$, and *expr*$_3$ (*expr*$_3$ must be nonzero when present) are any valid scalar integer expressions, and *name* is the optional construct name. The `do` and `end do` statements must be unnamed or bear the same *name*.

The number of iterations of a `do` construct is given by the formula

$$\texttt{max}\left(\frac{expr_2 - expr_1 + expr_3}{expr_3}, 0\right)$$

where `max` is a function which we shall meet in Section 9.3.2 and which here returns either the value of the first expression or zero, whichever is the larger. There is a consequence following from this definition, namely that if a loop begins with the statement

```
do  i = 1, n
```

then its body will not be executed at all if the value of `n` on entering the loop is zero or less. This is an example of the **zero-trip loop**, and results from the application of the `max` function.

A very simple form of the `do` statement is the unbounded

[name `:` *]* `do`

which specifies an endless loop. In practice, a means to exit from an endless loop is required, and this is provided in the form of the `exit` statement:

`exit` *[name]*

where *name* is optional and is used to specify from which `do` construct the exit should be taken in the case of nested constructs.[2] Execution of an `exit` statement causes control to be transferred to the next executable statement after the `end do` statement to which it refers. If no name is specified, it terminates execution of the innermost `do` construct in which it is enclosed. As an example of this form of `do`, suppose we have used the type `entry` of Section 2.12 to construct a chain of entries in a sparse vector, and we wish to find the entry with index 10, known to be present. If `first` points to the first entry, the code in Figure 4.5 is suitable.

Figure 4.5 Searching a linked list.

```
type(entry), pointer :: first, current
⋮
current => first
do
   if (current%index == 10) exit
   current => current%next
end do
```

The `exit` statement is also useful in a bounded loop when all iterations are not always needed.

A related statement is the `cycle` statement

[2] In fact, a named `exit` can be used to exit from nearly any construct, not just a loop; see Section 4.5.

```
cycle [ name ]
```

which transfers control to the `end do` statement of the corresponding construct. Thus, if further iterations are still to be carried out, the next one is initiated.

The value of a `do` construct index (if present) is incremented at the end of every loop iteration for use in the subsequent iteration. As the value of this index is available outside the loop after its execution, we have three possible situations, each illustrated by the following loop:

```
do  i = 1, n
   ⋮
   if (i==j) exit
   ⋮
end do
l = i
```

The situations are as follows:

i) If, at execution time, `n` has the value zero or less, `i` is set to 1 but the loop is not executed, and control passes to the statement following the `end do` statement.

ii) If `n` has a value which is greater than or equal to `j`, an exit will be taken at the `if` statement, and `l` will acquire the last value of `i`, which is of course `j`.

iii) If the value of `n` is greater than zero but less than `j`, the loop will be executed `n` times, with the successive values of `i` being $1, 2, \ldots,$ `n`. When reaching the end of the loop for the `n`th time, `i` will be incremented a final time, acquiring the value `n+1`, which will then be assigned to `l`.

We see how important it is to make careful use of loop indices outside the `do` block, especially when there is the possibility of the number of iterations taking on the boundary value of the maximum for the loop.

The `do` block, just mentioned, is the sequence of statements between the `do` statement and the `end do` statement. It is prohibited to jump into a `do` block or to its `end do` statement from anywhere outside the block.

It is similarly invalid for the block of a `do` construct (or any other construct, such as an `if` or `case` construct), to be only partially contained in a block of another construct. The construct must be completely contained in the block. The following two sequences are valid:

```
do i = 1, n
   if (scalar-logical-expr) then
      ⋮
   end if
end do
```

and

```
if (scalar-logical-expr) then
   do i = 1, n
```

```
      ⋮
   end do
else
   ⋮
end if
```

Any number of `do` constructs may be nested. We may thus write a matrix multiplication as shown in Figure 4.6.

Figure 4.6 Matrix multiplication as a triply nested `do` construct.

```
do  i = 1, n
   do  j = 1, m
      a(i,j) = 0.0
      do  l = 1, k
         a(i,j) = a(i,j) + b(i,l)*c(l,j)
      end do
   end do
end do
```

A further form of `do` construct, the `do concurrent` construct, is described in Section 7.17, and additional, but redundant, forms of `do` syntax in Appendix A.5.

Finally, it should be noted that many short `do` loops can be expressed alternatively in the form of array expressions and assignments. However, this is not always possible, and a particular danger to watch for is where one iteration of the loop depends upon a previous one. Thus, the loop

```
do i = 2, n
   a(i) = a(i-1) + b(i)
end do
```

cannot be replaced by the statement

```
a(2:n) = a(1:n-1) + b(2:n)       ! Beware
```

4.5 Exit from nearly any construct

The `exit` statement can, in fact, be used to complete the execution of any construct except the `do concurrent` construct (see Section 7.17). In order to do this, the construct must be named and that name used on the `exit` statement. An example of this is shown in Figure 4.7.

Note that an `exit` statement without a construct name exits the innermost `do` construct. Since the different behaviours can easily confuse, we recommend that if an `exit` from a non-`do` construct is used in proximity to an `exit` from a `do` construct (as in Figure 4.7), both `exit` statements have construct labels.

Note that it is prohibited to exit an outer construct from within a `critical` or `do concurrent` construct.

Figure 4.7 Exit from `if` construct.

```
adding_to_set: if (add_x_to_set) then
  find_position: do i=1, size(set)
    if (x==set(i)) exit adding_to_set
    if (x>set(i)) exit find_position
  end do find_position
  set = [set(:i-1), x, set(i:)]  !set is reallocated (see Section 6.7)
end if adding_to_set
```

4.6 The go to statement

Just occasionally, especially when dealing with error conditions, the control constructs that we have described may be inadequate for the programmer's needs. The remedy is to use the most disputed statement in programming languages – the `go to` statement – to **branch** to another statement. It is generally accepted that it is difficult to understand a program which is interrupted by many branches, especially if there is a large number of backward branches – those returning control to a statement preceding the branch itself.

The form of the unconditional `go to` statement is

`go to` *label*

where *label* is a statement label. This statement label must be present on an **executable statement** (a statement that can be executed, as opposed to one of an informative nature, like a declaration). An example is

```
  x = y + 3.0
  go to 4
3 x = x + 2.0
4 z = x + y
```

in which we note that after execution of the first statement, a branch is taken to the last statement, labelled 4. This is a **branch target statement**. The statement labelled 3 is jumped over, and can be executed only if there is a branch to the label 3 somewhere else. If the statement following an unconditional `go to` is unlabelled, it can never be reached and executed, thus is **dead code**, normally a sign of incorrect coding.

The statements within a block of a construct may be labelled, but the labels must never be referenced in such a fashion as to pass control into the range of a block from outside it, to an `else if` statement or to an `else` statement. It is permitted to pass control from a statement in a construct to the terminal statement of the construct, or to a statement outside its construct.

The `if` statement is normally used either to perform a single assignment depending on a condition, or to branch depending on a condition. The *action-stmt* may not be labelled separately. Examples are

```
if (flag) go to 6
if (x-y > 0.0) x = 0.0
```

Branching can also occur in an input/output statement (Chapter 10) with an `err=`, `end=`, or `err=` clause.

4.7 Summary

In this chapter we have introduced the four main features by which control in Fortran code may be programmed – the `if` statement and construct, the `case` construct, and the `do` construct. The `go to` statement has also been mentioned. The effective use of these features is a key to sound code.

We have touched upon the concept of a program unit as being like the chapter of a book. Just as a book may have only one chapter, so a complete program may consist of just one program unit, which is known as a main program. In its simplest form it consists of a series of statements of the kinds we have been dealing with so far, and terminates with an `end` statement, which acts as a signal to the computer to stop processing the current program. In order to test whether a program unit of this type works correctly, we need to be able to output, to a terminal or printer, the values of the computed quantities. This topic will be fully explained in Chapter 10, and for the moment we need to know only that this can be achieved by a statement of the form

```
print * , ' var1 = ', var1 , ' var2 = ', var2
```

which will output a line such as

```
var1 = 1.0  var2 = 2.0
```

Similarly, input data can be read by statements like

```
read *, val1, val2
```

This is sufficient to allow us to write simple programs like that in Figure 4.8, which outputs the converted values of a temperature scale between specified limits, and Figure 4.9, which constructs a linked list. Sample inputs are shown at the end of each example.

Figure 4.8 Print a conversion table.

```
!   Print a conversion table of the Fahrenheit and Celsius
!   temperature scales between specified limits.
!
   real      :: celsius, fahrenheit
   integer   :: low_temp, high_temp, temperature
   character :: scale
!
read_loop: do
!
!   Read scale and limits
      read *, scale, low_temp, high_temp
!
!   Check for valid data
      if (scale /= 'C' .and. scale /= 'F') exit read_loop
!
!   Loop over the limits
      do  temperature = low_temp, high_temp
!
!   Choose conversion formula
         select case (scale)
         case ('C')
            celsius = temperature
            fahrenheit = 9.0/5.0*celsius + 32.0
!   Print table entry
            print *, celsius, ' degrees C correspond to',   &
                fahrenheit, ' degrees F'
         case ('F')
            fahrenheit = temperature
            celsius = 5.0/9.0*(fahrenheit-32.0)
!   Print table entry
            print *, fahrenheit, ' degrees F correspond to',&
                celsius, ' degrees C'
         end select
      end do
   end do read_loop
!
!   Termination
   print *, ' End of valid data'
   end
C  90   100
F  20   32
*  0   0
```

Figure 4.9 Constructing and printing a linked list.

```
      type entry ! Type for sparse matrix
         real                 :: value
         integer              :: index
         type(entry), pointer :: next
      end type entry

      type(entry), pointer :: first, current
      integer              :: key
      real                 :: value
!
! Create a null list
      nullify (first)
!
! Fill the list
      do
         read *, key, value
         if (key <= 0) exit
         allocate (current)
         current = entry(value, key, first)
         first => current
      end do
!
! Print the list
      current => first
      do
         if (.not.associated(current)) exit
         print *, current%index, current%value
         current => current%next
      end do
   end

   1 4
   2 9
   0 0
```

Exercises

1. Write a program which

a) defines an array to have 100 elements;

b) assigns to the elements the values $1, 2, 3, \ldots, 100$;

c) reads two integer values in the range 1 to 100;

d) reverses the order of the elements of the array in the range specified by the two values.

2. The first two terms of the Fibonacci series are both 1, and all subsequent terms are defined as the sum of the preceding two terms. Write a program which reads an integer value `limit` and which computes and prints the values of the first `limit` terms of the series.

3. The coefficients of successive orders of the binomial expansion are shown in the normal Pascal triangle form as

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
etc.

Write a program which reads an integer value `limit` and prints the coefficients of the first `limit` lines of the Pascal triangle.

4. Define a character variable of length 80. Write a program which reads a value for this variable. Assuming that each character in the variable is alphabetic, write code which sorts them into alphabetic order and prints out the frequency of occurrence of each letter.

5. Write a program to read an integer value `limit` and print the first `limit` prime numbers, by any method.

6. Write a program which reads a value `x` and calculates and prints the corresponding value `x/(1.+x)`. The case `x` $= -1$. should produce an error message and be followed by an attempt to read a new value of `x`.

7. Given a chain of entries of the type `entry` of Section 2.12, modify the code in Figure 4.5 (Section 4.4) so that it removes the entry with index 10, and makes the previous entry point to the following entry.

5. Program units and procedures

5.1 Introduction

As we saw in the previous chapter, it is possible to write a complete Fortran program as a single unit, but it is preferable to break the program down into manageable units. Each such **program unit** corresponds to a program task that can be readily understood and, ideally, can be written, compiled, and tested in isolation. We will discuss three kinds of program unit, the main program, external subprogram, and module. Submodules obviate problems associated with very large modules and are discussed in Chapter 16.

A complete program must, as a minimum, include one **main program**. This may contain statements of the kinds that we have met so far in examples, but normally its most important statements are invocations or calls to subsidiary programs, each of which is known as a **subprogram**. A subprogram defines a **function** or a **subroutine**. These differ in that a function returns a single object and usually does not alter the values of its arguments (so that it represents a function in the mathematical sense), whereas a subroutine usually performs a more complicated task, returning several results through its arguments and by other means. Functions and subroutines are known collectively as **procedures**.

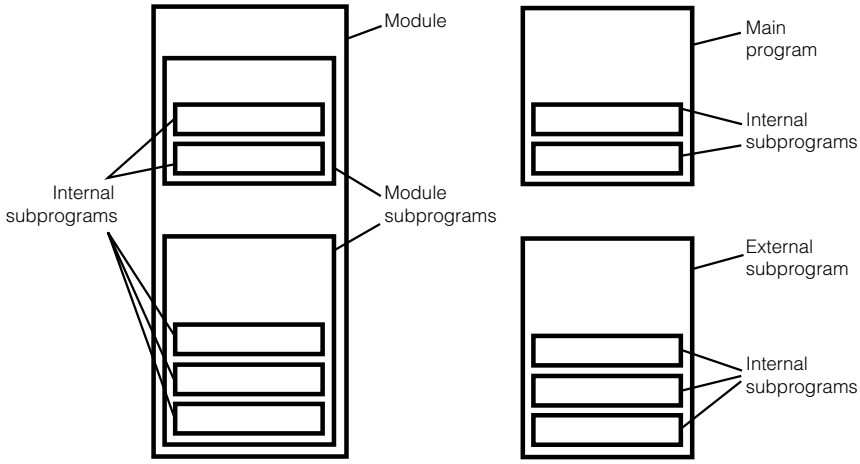
There are various kinds of subprograms. A subprogram may be a program unit in its own right, in which case it is called an **external subprogram** and defines an **external procedure**. External procedures may also be defined by means other than Fortran. A subprogram may be a member of a collection in a program unit called a **module**, in which case we call it a **module subprogram** and it defines a **module procedure**. A subprogram may be placed inside a module subprogram, an external subprogram, or a main program, in which case we call it an **internal subprogram** and it defines an **internal procedure**. Internal subprograms may not be nested, that is they may not contain further subprograms, and we expect them normally to be short sequences of code, say up to about twenty lines. We illustrate the nesting of subprograms in program units in Figure 5.1. If a program unit or subprogram contains a subprogram, it is called the **host** of that subprogram.

Besides containing a collection of subprograms, a module may contain data definitions, derived-type definitions, interface blocks (Section 5.11), and namelist groups (Section 8.20). This collection may provide facilities associated with some particular task, such as providing matrix arithmetic, a library facility, or a database. It may sometimes be large.

In this chapter we will describe program units and the statements that are associated with them. Within a complete program they may appear in any order, but many compilers require a module to precede other program units that use it.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

Figure 5.1 Nesting of subprograms in program units.



5.2 Main program

Every complete program must have one, and only one, main program. Optionally, it may contain calls to subprograms. A main program has the following form:

[`program` *program-name]*
    [specification-stmts]
    [executable-stmts]
[`contains`
    [internal-subprogram] . . .]
`end` *[* `program` *[program-name]]*

The `program` statement is optional, but we recommend its use. The *program-name* may be any valid Fortran name such as `model`. The only non-optional statement is the `end` statement, which has two purposes. It acts as a signal to the compiler that it has reached the end of the program unit and, when executed, it causes the program to stop.[1] If it includes *program-name*, this must be the name on the `program` statement. We recommend using the full form so that it is clear both to the reader and to the compiler exactly what is terminated by the `end` statement.

A main program without calls to subprograms is usually used only for short tests, as in

```
program test
   print *, 'Hello world!'
end program test
```

The specification statements define the environment for the executable statements. So far, we have met the type declaration statement (`integer`, `real`, `complex`, `logical`, `character`,

[1] For a coarray program (Chapter 17), the image stops executing but its data remain accessible to other images.

and `type`(*type-name*)) that specifies the type and other properties of the entities that it lists, and the type definition block (bounded by `type` *type-name* and `end type` statements). We will meet other specification statements in this and the next two chapters.

The executable statements specify the actions that are to be performed. So far, we have met the assignment statement, the pointer assignment statement, the `if` statement and construct, the `do` and `case` constructs, the `go to` statement, and the `read` and `print` statements. We will meet other executable statements in this and later chapters. Execution of a program always commences with the first executable statement of the main program.

The `contains` statement separates any internal subprograms from the body of the main program. We will describe internal subprograms in Section 5.6. They are excluded from the sequence of executable statements of the main program; if the last executable statement before a `contains` is executed without branching, the next statement executed will be the `end` statement. Note that although the syntax permits a `contains` statement without any following internal subprograms, it serves no purpose and so should be avoided. The `end` statement may be the target of a branch from one of the executable statements. If the `end` statement is executed, further execution stops.[1]

5.3 The stop statement

Another way to stop program execution is to execute a `stop` statement. This statement may appear in the main program or any subprogram. A well-designed program normally returns control to the main program for program termination, so the `stop` statement should appear there. However, in applications where several `stop` statements appear in various places in a complete program, it is possible to distinguish which of the `stop` statements has caused the termination by adding to each one a *stop code* consisting of a default integer or default character constant expression (Sections 2.6.1 and 2.6.4). This might be used by a given processor to indicate the origin of the `stop` in a message.[2] Examples are

```
stop
stop 12345
stop -2**20
stop 'Incomplete data. Program terminated.'
stop 'load_data_type_1'//': value out of range'
```

The Fortran standard requires that on termination by a `stop` statement, if any IEEE floating-point exception is signaling (Chapter 18), a warning message be written to the error file unit `error_unit` of the module `iso_fortran_env` (Section 9.24.1).

The standard also recommends that any stop code be written to the same unit and, if it is an integer, that it be used as the 'process exit status' if the operating system has such a concept. It further recommends that an exit status of zero be supplied if the stop code is of type character or the program is terminated by an `end program` statement. However, these are only recommendations, and in any case operating systems often have only a limited range for the process exit status, so values outside the range of $0-127$ should be avoided for this purpose.

[2]This statement is extended in Fortran 2018, see Section 23.5.

There is also an `error stop` statement (Section 17.14) that can be used for program termination. The main difference between the `stop` and `error stop` statements is that the latter causes **error termination** instead of normal termination. It has the same IEEE exception and stop code reporting requirements as `stop`, and the recommended process exit status with a numeric stop code is again the stop code value, but otherwise the exit status should be non-zero, in accordance with typical operating system conventions where zero indicates success and non-zero indicates failure. (This is also the recommended exit code for other error-termination situations, such as an unhandled input/output error or allocation failure.)

Other differences between normal and error termination are:

- normal termination properly closes all files, waiting for any input/output operation in progress to complete, but error termination has no such requirement (this could cause data loss if files are still being written);
- in a coarray program with multiple images (Chapter 17), the entire computation is terminated, not just a single image (Section 17.14).

Even though the normal and error termination exit code values are merely recommendations of the Fortran standard, it rarely makes sense to second-guess the processor's choices here. For these reasons, we recommend the use of an informative message rather than an integer for both the `stop` and `error stop` statements.

5.4 External subprograms

External subprograms are called from a main program or elsewhere, usually to perform a well-defined task within the framework of a complete program. Apart from the leading statement, they have a form that is very like that of a main program:

subroutine-stmt
    [specification-stmts]
    [executable-stmts]
[`contains`
    [internal-subprogram] . . .]
`end` *[* `subroutine` *[subroutine-name]]*

or

function-stmt
    [specification-stmts]
    [executable-stmts]
[`contains`
    [internal-subprogram] . . .]
`end` *[* `function` *[function-name]]*

The `contains` statement plays exactly the same role as within a main program (see Section 5.2). As before, although the syntax permits a `contains` statement without any following internal subprograms, we do not recommend doing that. The effect of executing an `end` statement in a subprogram is to return control to the caller, rather than to stop execution.

As for the `end program` statement, we recommend using the full form for the `end` statement so that it is clear both to the reader and to the compiler exactly what it terminates.

The simplest form of external subprogram defines a subroutine without any arguments and has a *subroutine-stmt* of the form

`subroutine` *subroutine-name*

Such a subprogram is useful when a program consists of a sequence of distinct phases, in which case the main program consists of a sequence of `call` statements that invoke the subroutines as in the example

```
program game         ! Main program to control a card game
   call shuffle      ! First shuffle the cards.
   call deal         ! Now deal them.
   call play         ! Play the game.
   call display      ! Display the result.
end program game     ! Cease execution.
```

But how do we handle the flow of information between the subroutines? How does `play` know which cards `deal` has dealt? There are, in fact, two methods by which information may be passed. The first is via data held in a module (Section 5.5) and accessed by the subprograms, and the second is via arguments (Section 5.7) in the procedure calls.

5.5 Modules

The third type of program unit, the module, provides a means of packaging global data, derived types and their associated operations, subprograms, interface blocks (Section 5.11), and namelist groups (Section 8.20). Everything associated with some task (such as interval arithmetic, see later in this section) may be collected into a module and accessed whenever it is needed. Those parts that are associated with the internal working and are of no interest to the user may be made 'invisible' to the user, which allows the internal design to be altered without the need to alter the program that uses it and prevents accidental alteration of internal data. Fortran libraries often consist of sets of modules.

The module has the form

`module` *module-name*
  [specification-stmts]
[`contains`
  [module-subprogram] ...]
`end` *[* `module` *[module-name]]*

As for other program units, although the syntax permits a `contains` statement without any following module subprograms, we do not recommend doing that. As for the `end program`, `end subroutine`, and `end function` statements, we recommend using the full form for the `end` statement.

In its simplest form, the body consists only of data specifications. For example

```
module state
   integer, dimension(52) :: cards
end module state
```

might hold the state of play of the game of Section 5.4. It is accessed by the statement

```
use state
```

appearing at the beginnings of the main program `game` and subprograms `shuffle`, `deal`, `play`, and `display`. The array `cards` is set by `shuffle` to contain the integer values 1 to 52 in a random order, where each integer value corresponds to a predefined playing card. For instance, 1 might stand for the ace of clubs, 2 for the two of clubs, etc. up to 52 for the king of spades. The array `cards` is changed by the subroutines `deal` and `play`, and finally accessed by subroutine `display`.

A further example of global data in a module would be the definitions of the values of the kind type parameters (Section 2.6) that might be required throughout a program. They can be placed in a module and used wherever they are required. On a processor that supports all the kinds listed, an example might be:

```
module numeric_kinds
   ! named constants for 4, 2, and 1 byte integers:
   integer, parameter ::                                &
        i4b = selected_int_kind(9),                     &
        i2b = selected_int_kind(4),                     &
        i1b = selected_int_kind(2)
   ! and for single, double and quadruple precision reals:
   integer, parameter ::                                &
        sp = kind(1.0),                                 &
        dp = selected_real_kind(2*precision(1.0_sp)),   &
        qp = selected_real_kind(2*precision(1.0_dp))
end module numeric_kinds
```

A very useful role for modules is to contain definitions of types and their associated operators. For example, a module might contain the type `interval` of Section 3.9, as shown in Figure 5.2. Given this module, any program unit needing this type and its operators need only include the statement

```
use interval_arithmetic
```

at the head of its specification statements.

A module subprogram has exactly the same form as an external subprogram. It has access to other entities of the module, including the ability to call other subprograms of the module, rather as if it contained a `use` statement for its module.

A module may contain `use` statements that access other modules. It must not access itself directly or indirectly through a chain of `use` statements, for example `a` accessing `b` and `b` accessing `a`. No ordering of modules is required by the standard, but normal practice is to require each module to precede its use. We recommend this practice, which is required by many compilers.

It is possible within a module to specify that some of the entities are private to it and cannot be accessed from other program units. Also, there are forms of the `use` statement that allow access to only part of a module and forms that allow renaming of the entities accessed. These features will be explained in Sections 8.6.1 and 8.14. For the present, we assume that the whole module is accessed without any renaming of the entities in it.

Figure 5.2 A module for interval arithmetic.

```
module interval_arithmetic
   type interval
      real :: lower, upper
   end type interval
   interface operator(+)
      module procedure add_intervals
   end interface
   ⋮
contains
   function add_intervals(a,b)
      type(interval)             :: add_intervals
      type(interval), intent(in) :: a, b
      add_intervals%lower = a%lower + b%lower
      add_intervals%upper = a%upper + b%upper
   end function add_intervals
   ⋮
end module interval_arithmetic
```

5.6 Internal subprograms

We have seen that internal subprograms may be defined inside main programs and external subprograms, and within module subprograms. They have the form

subroutine-stmt
    [specification-stmts]
    [executable-stmts]
`end` *[* `subroutine` *[subroutine-name]]*

or

function-stmt
    [specification-stmts]
    [executable-stmts]
`end` *[* `function` *[function-name]]*

that is, the same form as a module subprogram, except that they may not contain further internal subprograms. An internal subprogram automatically has access to all the host's entities, including the ability to call its other internal subprograms. Internal subprograms must be preceded by a `contains` statement in the host. As for other `end` statements, we recommend using the full form of the `end` statement for internal subprograms.

In the rest of this chapter we describe several properties of subprograms that apply to external, module, and internal subprograms. We therefore do not need to describe internal subprograms separately. An example is given in Figure 5.12 (Section 5.15).

5.7 Arguments of procedures

Procedure arguments provide an alternative means for two program units to access the same data. Returning to our card game example, instead of placing the array `cards` in a module, we might declare it in the main program and pass it as an actual argument to each subprogram, as shown in Figure 5.3.

Figure 5.3 Subroutine calls with actual arguments.

```
program game       ! Main program to control a card game
   integer, dimension(52) :: cards
   call shuffle(cards)       ! First shuffle the cards.
   call deal(cards)          ! Now deal them.
   call play(cards)          ! Play the game.
   call display(cards)       ! Display the result.
end program game             ! Cease execution.
```

Each subroutine receives `cards` as a dummy argument. For instance, `shuffle` has the form shown in Figure 5.4.

Figure 5.4 A subroutine with a dummy argument.

```
subroutine shuffle(cards)
   ! Subroutine that places the values 1 to 52 in cards
   ! in random order.
   integer, dimension(52) :: cards
   ! Statements that fill cards
   ⋮
end subroutine shuffle   ! Return to caller.
```

We can, of course, imagine a card game in which `deal` is going to deal only three cards to each of four players. In this case, it would be a waste of time for `shuffle` to prepare a deck of 52 cards when only the first 12 cards are needed. This can be achieved by requesting `shuffle` to limit itself to a number of cards that is transmitted in the calling sequence thus:

```
call shuffle(3*4, cards(1:12))
```

Inside `shuffle`, we would define the array to be of the given length and the algorithm to fill `cards` would be placed in a `do` construct with this number of iterations, as seen in Figure 5.5.

We have seen how it is possible to pass an array and a constant expression between two program units. An actual argument may be any variable or expression (or a procedure name, see Section 5.12). Each dummy argument of the called procedure must agree with the corresponding actual argument in type, type parameters, and shape.[3] However, the names do not have to be the same. For instance, if two decks had been needed, we might have written the code thus:

[3]The requirements on character length and shape agreement are relaxed in Section 19.5.

Figure 5.5 A subroutine with two dummy arguments.

```
subroutine shuffle(ncards, cards)
   integer                    :: ncards, icard
   integer, dimension(ncards) :: cards
   do icard = 1, ncards
      ⋮
      cards(icard) = ...
   end do
end subroutine shuffle
```

```
program game
   integer, dimension(52) :: acards, bcards
   call shuffle(acards)        ! First shuffle the a deck.
   call shuffle(bcards)        ! Next shuffle the b deck.
   ⋮
end program game
```

The important point is that subprograms can be written independently of one another, the association of the dummy arguments with the actual arguments occurring each time the call is executed. This is known as **argument association**. We can imagine `shuffle` being used in other programs which use other names. In this manner, libraries of subprograms may be built up.

Being able to have different names for actual and dummy arguments provides useful flexibility, but it should only be used when it is actually needed. When the same name can be used, the code is more readable.

As the type of an actual argument and its corresponding dummy argument must agree, care must be taken when using component selection within an actual argument. Thus, supposing the derived-type definitions `point` and `triangle` of Figure 2.1 (Section 2.9) are available in a module `def`, we might write

```
      use def
      type(triangle) :: t
      ⋮
      call sub(t%a)
      ⋮
  contains
      subroutine sub(p)
         type(point) :: p
      ⋮
```

A dummy argument of a procedure may be used as an actual argument in a procedure call. The called procedure may use its dummy argument as an actual argument in a further

procedure call. A chain is built up with argument association at every link. The chain ends at an object that is not a dummy argument, which is known as the **ultimate argument** of the original dummy argument.

5.7.1 Assumed-shape arrays

Outside Appendix B, we require that the shapes of actual and dummy arguments agree, and so far we have achieved this by passing the extents of the array arguments as additional arguments. However, it is possible to require that the shape of the dummy array be taken automatically to be that of the corresponding actual array argument. Such an array is said to be an **assumed-shape array**. When the shape is declared by the `dimension` clause, each dimension has the form

[lower-bound] :

where *lower-bound* is an integer expression that may depend on module data or the other arguments (see Section 8.18 for the exact rules). If *lower-bound* is omitted, the default value is one. Note that it is the shape that is passed, and not the upper and lower bounds. For example, if the actual array is `a`, declared thus:

```
real, dimension(0:10, 0:20) :: a
```

and the dummy array is `da`, declared thus:

```
real, dimension(:, :) :: da
```

then `a(i,j)` corresponds to `da(i+1,j+1)`; to get the natural correspondence, the lower bound must be declared:

```
real, dimension(0:, 0:) :: da
```

In order that the compiler knows that additional information is to be supplied, the interface must be explicit (Section 5.11) at the point of call. A dummy array with the `pointer` or `allocatable` attribute is not regarded as an assumed-shape array because its shape is not necessarily assumed.

5.7.2 Pointer arguments

A dummy argument is permitted to have the attribute `pointer`. In this case, the actual argument must also have the attribute `pointer`. When the subprogram is invoked, the rank of the actual argument must match that of the dummy argument, and its pointer association status is passed to the dummy argument. On return, the actual argument normally takes its pointer association status from that of the dummy argument, but it becomes undefined if the dummy argument is associated with a target that becomes undefined when the return is executed (for example, if the target is a local variable that does not have the `save` attribute, Section 8.10).

In the case of a module or internal procedure, the compiler knows when the dummy argument is a pointer. In the case of an external or dummy procedure, the compiler assumes that the dummy argument is not a pointer unless it is told otherwise in an interface block (Section 5.11).

A pointer actual argument is also permitted to correspond to a non-pointer dummy argument. In this case, the pointer must have a target and the target is associated with the dummy argument, as in

```
      real, pointer :: a(:,:)
      ⋮
      allocate ( a(80,80) )
      call find (a)
      ⋮
   subroutine find (c)
      real :: c(:,:) ! Assumed-shape array
```

The term **effective argument** is used to refer to the entity that is associated with a dummy argument, that is, the actual argument if it is not a pointer, and its target otherwise.

5.7.3 Restrictions on actual arguments

There are two important restrictions on actual arguments, which are designed to allow the compiler to optimize on the assumption that the dummy arguments are distinct from each other and from other entities that are accessible within the procedure. For example, a compiler may arrange for an array to be copied to a local variable on entry, and copied back on return. While an actual argument is associated with a dummy argument the following statements hold:

i) Action that affects the allocation status or pointer association status of the argument or any part of it (any pointer assignment, allocation, deallocation, or nullification) must be taken through the dummy argument. If this is done, then throughout the execution of the procedure, the argument may be referenced only through the dummy argument.

ii) Action that affects the value of the argument or any part of it must be taken through the dummy argument unless

   a) the dummy argument has the `pointer` attribute;

   b) the part is all or part of a pointer subobject; or

   c) the dummy argument has the `target` attribute, the dummy argument does not have intent `in` (Section 5.9), the dummy argument is scalar or an assumed-shape array (Section 5.7.1), and the actual argument is a target other than an array section with a vector subscript.

   If the value of the argument or any part of it is affected through a dummy argument for which neither a), b), nor c) holds, then throughout the execution of the procedure, the argument may be referenced only through that dummy argument.

An example of i) is a pointer that is nullified (Section 3.14) while still associated with the dummy argument. As an example of ii), consider

```
call modify(a(1:5), a(3:9))
```

Here, `a(3:5)` may not be changed through either dummy argument since this would violate the rule for the other argument. However, `a(1:2)` may be changed through the first argument and `a(6:9)` may be changed through the second. Another example is an actual argument that is an object being accessed from a module; here, the same object must not be accessed from the module by the procedure and redefined. As a third example, suppose an internal procedure call associates a host variable `h` with a dummy argument `d`. If `d` is defined during the call, then at no time during the call may `h` be referenced directly.

5.7.4 Arguments with the target attribute

In most circumstances an implementation is permitted to make a copy of an actual argument on entry to a procedure and copy it back on return. This may be desirable on efficiency grounds, particularly when the actual argument is not held in contiguous storage. In any case, if a dummy argument has neither the `target` nor `pointer` attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument but remain associated with the actual argument.

However, copy-in copy-out is not allowed when

i) a dummy argument has the `target` attribute and is either scalar or is an assumed-shaped array; and

ii) the actual argument is a target other than an array section with a vector subscript.

In this case, the dummy and actual arguments must have the same shape, any pointer associated with the actual argument becomes associated with the dummy argument on invocation, and any pointer associated with the dummy argument on return remains associated with the actual argument.

When a dummy argument has the `target` attribute, but the actual argument is not a target or is an array section with a vector subscript, any pointer associated with the dummy argument obviously becomes undefined on return.

In other cases where the dummy argument has the `target` attribute, whether copy-in copy-out occurs is processor dependent. No reliance should be placed on the pointer associations with such an argument after the invocation.

5.8 The return statement

We saw in Section 5.2 that if the `end` statement of a main program is executed, further execution stops. Similarly, if the `end` statement in a subprogram is executed, control returns to the point of invocation. Just as the `stop` statement is an executable statement that provides an alternative means of stopping execution, so the `return` statement provides an alternative means of returning control from a subprogram. It has the form

```
return
```

and must not appear in a main program.

5.9 Argument intent

In Figure 5.5, the dummy argument `cards` was used to pass information out from `shuffle` and the dummy argument `ncards` was used to pass information in; a third possibility is for a dummy argument to be used for both input and output variables. We can specify such intent on the type declaration statement for the argument, for example:

```
subroutine shuffle(ncards, cards)
   integer, intent(in)                    :: ncards
   integer, intent(out), dimension(ncards) :: cards
```

For input/output arguments, intent `inout` may be specified.

If a dummy argument is specified with intent `in`, it (or any part of it) must not be redefined by the procedure, say by appearing on the left-hand side of an assignment or by being passed on as an actual argument to a procedure that redefines it. For the specification intent `inout`, the corresponding actual argument must be a variable because the expectation is that it will be redefined by the procedure. For the specification intent `out`, the corresponding actual argument must again be a variable; in this case, the intention is that it be used only to pass information out, so it becomes undefined on entry to the procedure, apart from any components with default initialization (Section 8.5.5).

If a function specifies a defined operator (Section 3.9), the dummy arguments must have intent `in`. If a subroutine specifies defined assignment (Section 3.10), the first argument must have intent `out` or `inout`, and the second argument must have intent `in`.

If a dummy argument has no intent, the actual argument may be a variable or an expression, but the actual argument must be a variable if the dummy argument is redefined. In this context we note that an actual argument variable, say `x`, is transformed into an expression if it is enclosed in parentheses, `(x)`; its value is then passed and cannot be redefined. The compiler will need to make a copy if the variable `x` could be changed during execution of the procedure, e.g. if `x` is not a local variable or is accessible via a pointer. We recommend that all dummy arguments be given a declared intent, allowing compilers to make more checks at compile time and as good documentation.

If a dummy argument has the `pointer` attribute, any intent refers to its pointer association (and **not** to the value of the target); that is, it refers to the descriptor. An intent `out` pointer has undefined association status on entry to the procedure; an intent `in` pointer cannot be nullified or associated during execution of the procedure; and the actual argument for an intent `inout` pointer must be a pointer variable (that is, it cannot be a reference to a pointer-valued function).

Note that, although an intent `in` pointer cannot have its pointer association status changed inside the procedure, if it is associated with a target the value of its target may be changed. For example,

```
subroutine maybe_clear(p)
   real, pointer, intent(in) :: p(:)
   if (associated(p)) p = 0.0
end subroutine maybe_clear
```

Likewise, if a dummy argument is of a derived type with a pointer component, its `intent` attribute refers to the pointer association status of that component (and **not** to the target

of the component). For example, if the intent is `in`, no pointer assignment, allocation, or deallocation is permitted.

5.10 Functions

Functions are similar to subroutines in many respects, but they are invoked within an expression and return a value that is used within the expression. For example, the subprogram in Figure 5.6 returns the distance between two points in space; the statement

```
if (distance(a, c) > distance(b, c) ) then
```

invokes the function twice in the logical expression that it contains.

Figure 5.6 A function that returns the distance between two points in space. The intrinsic function `sqrt` is defined in Section 9.4.

```
function distance(p, q)
   real                              :: distance
   real, intent(in), dimension(3) :: p, q
   distance = sqrt( (p(1)-q(1))**2 + (p(2)-q(2))**2 +    &
                    (p(3)-q(3))**2 )
end function distance
```

Note the type declaration for the function result. The result behaves just like a dummy argument with intent `out`. It is initially undefined, but once defined it may appear in an expression and it may be redefined. The type may also be defined on the `function` statement:

```
real function distance(p, q)
```

It is permissible to write functions that change the values of their arguments, modify values in modules, rely on local data saved (Section 8.10) from a previous invocation, or perform input/output operations. However, these are known as **side-effects** and conflict with good programming practice. Where they are needed, a subroutine should be used. It is reassuring to know that when a function is called, nothing else goes on 'behind the scenes', and it may be very helpful to an optimizing compiler, particularly for internal and module subprograms. A formal mechanism for avoiding side-effects is provided, but we defer its description to Section 7.8.

A function result may be an array, in which case it must be declared as such.

A function result may also be a pointer. The result is initially undefined. Within the function, it must become associated or defined as disassociated. We expect the function reference usually to be such that a pointer assignment takes place for the result, that is, the reference occurs as the right-hand side of a pointer assignment (Section 3.13), for example,

```
real          :: x(100)
real, pointer :: y(:)
⋮
y => compact(x)
⋮
```

or as a pointer component of a structure constructor. The reference may also occur as a primary of an expression or as the right-hand side of an ordinary assignment, in which case the result must become associated with a target that is defined and the value of the target is used. We do not recommend this practice, however, since it is likely to lead to memory leakage, discussed at the end of Section 6.6.

The value returned by a non-pointer function must always be defined.

As well as being a scalar or array value of intrinsic type, a function result may also be a scalar or array value of a derived type, as we have seen already in Section 3.9. When the function is invoked, the function value must be used as a whole, that is, it is not permitted to be qualified by substring, array-subscript, array-section, or structure-component selection.

Although this is not very useful, a function is permitted to have an empty argument list. In this case, the brackets are obligatory both within the function statement and at every invocation.

5.10.1 Prohibited side-effects

In order to assist an optimizing compiler, the standard prohibits reliance on certain side-effects. It specifies that it is not necessary for a processor to evaluate all the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise. For example, in evaluating

```
x>y .or. l(z)  ! x, y, and z are real; l is a logical function
```

the function reference need not be made if `x` is greater than `y`. Since some processors will make the call and others will not, any variable (for example `z`) that is redefined by the function is regarded as undefined following such an expression evaluation. Similarly, it is not necessary for a processor to evaluate any subscript or substring expressions for an array of zero size or character object of zero character length.

Another prohibition is that a function reference must not redefine the value of a variable that appears in the same statement or affect the value of another function reference in the same statement. For example, in

```
d = max(distance(p,q), distance(q,r))
```

`distance` is required not to redefine its arguments. This rule allows any expressions that are arguments of a single procedure call to be evaluated in any order. With respect to this rule, an `if` statement,

`if (`*lexpr*`)` *stmt*

is treated as the equivalent `if` construct

```
if (lexpr) then
   stmt
end if
```

and the same is true for the `where` statement (Section 7.6).

5.11 Explicit and implicit interfaces

A call to an internal subprogram must be from a statement within the same program unit. It may be assumed that the compiler will process the program unit as a whole and will therefore

know all about any internal subprogram. In particular, it will know about its **interface**, that is, whether it defines a function or a subroutine, the names and properties of the arguments, and the properties of the result if it defines a function. This, for example, permits the compiler to check whether the actual and dummy arguments match in the way that they should. We say that the interface is **explicit**.

A call to a module subprogram must either be from another statement in the module or from a statement following a `use` statement for the module. In both cases the compiler will know all about the subprogram, and again we say that the interface is explicit. Similarly, intrinsic procedures (Chapter 9) always have explicit interfaces.

When compiling a call to an external or dummy procedure (Section 5.12), the compiler normally does not have a mechanism to access its code. We say that the interface is **implicit**. All the compiler has is the information about the interface that is implicit in the statements in the environment of the invocation, for example the number of arguments and their types. To specify that a name is that of an external or dummy procedure, the `external` statement is available. It has the form

`external` *external-name-list*

and appears with other specification statements, after any `use` or `implicit` statements (Section 8.2) and before any executable statements. The type and type parameters of a function with an implicit interface are usually specified by a type declaration statement for the function name; an alternative is by the rules of implicit typing (Section 8.2) applied to the name, but this is not available in a module unless the function has the `private` attribute (see Section 8.6.1).

The `external` statement merely specifies that each *external-name* is the name of an external or dummy procedure. It does not specify the interface, which remains implicit. However, a mechanism is provided for the interface to be specified. It may be done through an interface block of the form

```
interface
   interface-body
end interface
```

Normally, the *interface-body* is an exact copy of the subprogram's header, the specifications of its arguments and function result, and its `end` statement. However,

- the names of the arguments may be changed;
- other specifications may be included (for example, for a local variable), but not internal procedures, `data` statements, or `format` statements;
- the information may be given by a different combination of statements;[4]
- in the case of an array argument or function result, the expressions that specify a bound may differ as long as their values can never differ; and

[4] A practice that is permitted by the standard, but which we do not recommend, is for a dummy argument to be declared implicitly as a procedure by invoking it in an executable statement. If the subprogram has such a dummy procedure, the interface will need an `external` statement for that dummy procedure.

- a recursive procedure (Sections 5.16 and 5.17) or a pure procedure (Section 7.8) need not be specified as such if it is not called as such.

An *interface-body* may be provided for a call to an external procedure defined by means other than Fortran (usually C or assembly language).

Naming a procedure in an `external` statement or giving it an interface body (doing both is not permitted) ensures that it is an external or dummy procedure. We strongly recommend the practice for external procedures, since otherwise the processor is permitted to interpret the name as that of an intrinsic procedure. It is needed for portability since processors are permitted to provide additional intrinsic procedures. Naming a procedure in an `external` statement makes all versions of an intrinsic procedure having the same name unavailable. The same is true for giving it an interface body (but not when the interface is generic, Section 5.18).

The interface block is placed in a sequence of specification statements and this suffices to make the interface explicit. Perhaps the most convenient way to do this is to place the interface block among the specification statements of a module and then `use` the module. Libraries can be written as sets of external subprograms together with modules holding interface blocks for them. This keeps the modules of modest size. Note that if a procedure is accessible in a scoping unit, its interface is either explicit or implicit there. An external procedure may have an explicit interface in some scoping units and an implicit interface in others.

Interface blocks may also be used to allow procedures to be called as defined operators (Section 3.9), as defined assignments (Section 3.10), or under a single generic name. We therefore defer description of the full generality of the interface block until Section 5.18, where overloading is discussed.

An explicit interface is required to invoke a procedure with a pointer or target dummy argument or a pointer function result, and is required for several useful features that we will meet later in this and the next chapter. It is needed so that the processor can make the appropriate linkage. Even when not strictly required, it gives the compiler an opportunity to examine data dependencies and thereby improve optimization. Explicit interfaces are also desirable because of the additional security that they provide. It is straightforward to ensure that all interfaces are explicit, and we recommend the practice.

5.11.1 The import statement

As we have seen, an interface body does not access its environment by host association, and therefore cannot use any named constants and derived types defined therein. In particular, it might be desirable in a module procedure to be able to describe a dummy procedure that uses types and kind type parameters defined in the module, but without host association this is impossible, as a module cannot 'use' itself. This problem is solved by the `import` statement. This statement can be used only in an interface body, and gives access to named entities of the containing scoping unit.

For example, in Figure 5.7 the interface body would be invalid without the `import` statement shown in bold, because it would have no means of accessing either type `t` or the constant `wp`.

Figure 5.7 Using an `import` statement to provide an explicit interface using types and constants defined in the module.

```
module m
   integer, parameter :: wp = kind(0.0d0)
   type t
      ⋮
   end type t
contains
   subroutine apply(fun,...)
      interface
         function fun(f)
            import  :: t, wp
            type(t)  :: fun
            real(wp) :: f
         end function fun
      end interface
   end subroutine apply
end module m
```

The statement must be placed after any `use` statements but ahead of any other statements of the body. It has the general form

`import` *[[*`::`*] import-name-list]*

where each *import-name* is that of an entity that is accessible in the containing scoping unit.[5] If an imported entity is defined in the containing scoping unit, it must be explicitly declared prior to the interface body.

An `import` statement without a list imports all entities from the containing scoping unit that are not declared to be local entities of the interface body; this works the same way as normal host association.

5.12 Procedures as arguments

So far, we have taken the actual arguments of a procedure invocation to be variables and expressions, but another possibility is for them to be procedures. Let us consider the case of a library subprogram for function minimization. It needs to receive the user's function, just as the subroutine `shuffle` in Figure 5.5 needs to receive the required number of cards. The minimization code might look like the code in Figure 5.8. Notice the way the procedure argument is declared by an interface block playing a similar role to that of the type declaration statement for a data object.

Just as the type and shape of actual and dummy data objects must agree, so must the properties of the actual and dummy procedures. The agreement is exactly as for a procedure

[5] This statement is extended in Fortran 2018, see Section 23.11.

and an interface body for that procedure (see Section 5.11). It would make no sense to specify an `intent` attribute (Section 5.9) for a dummy procedure, and this is not permitted.

Figure 5.8 A library subprogram for function minimization.

```
real function minimum(a, b, func) ! Returns the minimum
        ! value of the function func(x) in the interval (a,b)
   real, intent(in) :: a, b
   interface
      real function func(x)
         real, intent(in) :: x
      end function func
   end interface
   real :: f,x
   ⋮
   f = func(x)   ! invocation of the user function.
   ⋮
end function minimum
```

Figure 5.9 Invoking the library code of Figure 5.8.

```
module code
contains
   real function fun(x)
      real, intent(in) :: x
      ⋮
   end function fun
end module code
program main
   use code
   real :: f
   ⋮
   f = minimum(1.0, 2.0, fun)
   ⋮
end program main
```

On the user side, the code may look like that in Figure 5.9. Notice that the structure is rather like a sandwich: user-written code invokes the minimization code which in turn invokes user-written code. An external procedure here would instead require the presence of an interface block and reference to an abstract interface (Section 14.1), or, as a very minimum, the procedure name would have to be declared in an `external` statement.

The procedure that is passed can be an external, internal, or module procedure, or an associated procedure pointer (Section 14.2). The actual argument may be a generic procedure name (Section 5.18); if it is also a **specific name**, only the specific procedure is passed.

When an internal procedure is passed as an actual argument, the environment of the host procedure is passed with it. That is, when it is invoked via the corresponding dummy argument, it has access to the variables of the host procedure as if it had been invoked there. For example, in Figure 5.10, invocations of the function `fun` from `integrate` will use the values for the variables `freq` and `alpha` from the host procedure.

Figure 5.10 Quadrature using internal procedures.

```
subroutine s(freq, alpha, lower, upper, ...)
  real(wp), intent(in) :: freq, alpha, lower, upper
  ⋮
  z = integrate(fun, lower, upper)
  ⋮
contains
  real(wp) function fun(x)
    real(wp), intent(in) :: x
    fun = x*sin(freq*x)/sqrt(1-alpha*x**2)
  end function
end subroutine
```

If the host procedure is recursive (Sections 5.16 and 5.17), the instance that called the procedure with its internal procedure as an actual argument is known as the **host instance** of the internal procedure. It is the data in this instance that can be accessed by the internal procedure.

Apart from the convenience, this code can in principle safely be part of a multi-threaded program because the data for the function evaluation are not being passed by global variables.

5.13 Keyword and optional arguments

In practical applications, argument lists can get long and actual calls may need only a few arguments. For example, a subroutine for constrained minimization might have the form

```
subroutine mincon(n, f, x, upper, lower,                    &
                  equalities, inequalities, convex, xstart)
```

For many calls there may be no upper bounds, or no lower bounds, or no equalities, or no inequalities, or it may not be known whether the function is convex, or a sensible starting point may not be known. All the corresponding dummy arguments may be declared `optional` (see also Section 8.9). For instance, the bounds might be declared by the statement

```
real, optional, dimension(n) :: upper,lower
```

If the first four arguments are the only wanted ones, we may use the statement

```
call mincon(n, f, x, upper)
```

but usually the wanted arguments are scattered. In this case, we may follow a (possibly empty) ordinary positional argument list for leading arguments by a keyword argument list, as in the statement

```
call mincon(n, f, x, equalities=q, xstart=x0)
```

The keywords are the dummy argument names and there must be no further positional arguments after the first keyword argument.

This example also illustrates the merits of both positional and keyword arguments as far as readability is concerned. A small number of leading positional arguments (for example, `n`, `f`, and `x`) are easily linked in the reader's mind to the corresponding dummy arguments. Beyond this, the keywords are very helpful to the reader in making these links. We recommend their use for long argument lists even when there are no gaps caused by optional arguments that are not present.

A non-optional argument must appear exactly once, either in the positional list or in the keyword list. An optional argument may appear at most once, either in the positional list or in the keyword list. An argument must not appear in both lists.

The called subprogram needs some way to detect whether an argument is present so that it can take appropriate action when it is not. This is provided by the intrinsic function `present` (see Section 9.2). For example

```
present(xstart)
```

returns the value `.true.` if the current call has provided a starting point and `.false.` otherwise. When it is absent, the subprogram might, for example, use a random number generator to provide a starting point.

A slight complication occurs if an optional dummy argument is used within the subprogram as an actual argument in a procedure invocation. For example, our minimization subroutine might start by calling a subroutine that handles the corresponding equality problem by the call

```
call mineq(n, f, x, equalities, convex, xstart)
```

In such a case, an absent optional argument is also regarded as absent in the second-level subprogram. For instance, when `convex` is absent in the call of `mincon`, it is regarded as absent in `mineq` too. Such absent arguments may be propagated through any number of calls, provided the dummy argument is optional in each case. An absent argument further supplied as an actual argument must be specified as a whole, and not as a subobject. Furthermore, an absent pointer is not permitted to be associated with a non-pointer dummy argument (the target is doubly absent).

Since the compiler will not be able to make the appropriate associations unless it knows the keywords (dummy argument names), the interface must be explicit (Section 5.11) if any of the dummy arguments are optional or keyword arguments are in use. Note that an interface block may be provided for an external procedure to make the interface explicit. In all cases where an interface block is provided, it is the names of the dummy arguments in the block that are used to resolve the associations.

5.14 Scope of labels

Execution of the main program or a subprogram always starts at its first executable statement and any branching always takes place from one of its executable statements to another. Indeed, each subprogram has its own independent set of labels. This includes the case of a host subprogram with several internal subprograms. The same label may be used in the host and the internal subprograms without ambiguity.

This is our first encounter with **scope**. The scope of a label is a main program or a subprogram, excluding any internal subprograms that it contains. The label may be used unambiguously anywhere among the executable statements of its scope. Notice that the host `end` statement may be labelled and be a branch target from a host statement; that is, the internal subprograms leave a hole in the scope of the host (see Figure 5.11).

Figure 5.11 An example of nested scopes.

```
module scope1                ! scope 1
  ⋮                          ! scope 1
contains                     ! scope 1
   subroutine scope2         ! scope 2
      type scope3            ! scope 3
        ⋮                    ! scope 3
      end type scope3        ! scope 3
      interface              ! scope 2
        ⋮                    ! scope 4
      end interface          ! scope 2
     ⋮                       ! scope 2
   contains                  ! scope 2
      function scope5(...)   ! scope 5
        ⋮                    ! scope 5
      end function scope5    ! scope 5
   end subroutine scope2     ! scope 2
end module scope1            ! scope 1
```

5.15 Scope of names

In the case of a named entity, there is a similar set of statements within which the name may always be used to refer to the entity. Here, derived-type definitions and interface blocks as well as subprograms can knock holes in scopes. This leads us to regard each program unit as consisting of a set of non-overlapping scoping units. A **scoping unit** is one of the following:

- a derived-type definition;
- a procedure interface body, excluding any derived-type definitions and interface bodies contained within it; or

- a program unit or subprogram, excluding derived-type definitions, interface bodies, and subprograms contained within it.

An example containing five scoping units is shown in Figure 5.11.

Once an entity has been declared in a scoping unit, its name may be used to refer to it in that scoping unit. An entity declared in another scoping unit is always a different entity even if it has the same name and exactly the same properties.[6] Each is known as a **local entity**. This is very helpful to the programmer, who does not have to be concerned about the possibility of accidental name clashes. Note that this is true for derived types, too. Even if two derived types have the same name and the same components, entities declared with them are treated as being of different types.[6]

A `use` statement of the form

`use` *module-name*

is regarded as a redeclaration of all the module entities inside the local scoping unit, with exactly the same names and properties. The module entities are said to be accessible by **use association**. Names of entities in the module may not be used to declare local entities (but see Section 8.14 for a description of further facilities provided by the `use` statement when greater flexibility is required).

Figure 5.12 Examples of host association.

```
subroutine outer
   real :: x, y
   ⋮
contains
   subroutine inner
      real :: y
      y = f(x) + 1. ! x and f accessed by host association
      ⋮
   end subroutine inner
   function f(z)
      real             :: f
      real, intent(in) :: z
      ⋮
   end function f
end subroutine outer
```

In the case of a derived-type definition, a module subprogram, or an internal subprogram, the scoping unit that immediately contains it is known as the **host scoping unit**. The name of an entity in the host scoping unit (including an entity accessed by use association) is treated as being automatically redeclared with the same properties, provided no entity with this name is declared locally, is a local dummy argument or function result, or is accessed by

[6] Apart from the effect of storage association, which is not discussed until Appendix A and whose use we strongly discourage.

use association. The host entity is said to be accessible by **host association**. For example, in the subroutine `inner` of Figure 5.12, `x` is accessible by host association, but `y` is a separate local variable and the `y` of the host is inaccessible. We note that `inner` calls another internal procedure that is a function, `f`; it must not contain a type specification for that function, as the interface is already explicit. Such a specification would, in fact, declare a different, external function of that name. The same remark applies to a module procedure calling a function in the same module.

Note that the host has no access to the local entities of a subroutine that it contains.

Host association does not extend to interface blocks unless an `import` statement (Section 5.11.1) is used. This allows an interface body to be constructed mechanically from the specification statements of an external procedure.

Within a scoping unit, each named data object, procedure, derived type, named construct, and namelist group (Section 8.20) must have a distinct name, with the one exception of generic names of procedures (to be described in Section 5.18). Note that this means that any appearance of the name of an intrinsic procedure in another role makes the intrinsic procedure inaccessible by its name (the renaming facility described in Section 8.14 allows an intrinsic procedure to be accessed from a module and renamed). Within a derived-type definition, each component of the type, each intrinsic procedure referenced, and each derived type or named constant accessed by host association, must have a distinct name. Apart from these rules, names may be reused. For instance, a name may be used for the components of two types, or the arguments of two procedures referenced with keyword calls.

The names of program units and external procedures are **global**, that is available anywhere in a complete program. Each must be distinct from the others and from any of the local entities of the program unit.

At the other extreme, the `do` variable of an implied-`do` in a `data` statement (Section 8.5.2) or an array constructor (Section 7.16) has a scope that is just the implied-`do`. It is different from any other entity with the same name.

5.16 Direct recursion

A subprogram that invokes itself, either directly or indirectly through a sequence of other invocations, is required to have the keyword `recursive` prefixed to its leading statement.[7] Where the subprogram is a function that calls itself directly in this fashion, the function name cannot be used for the function result and another name is needed. This is done by adding a further clause to the `function` statement as in Figure 5.13, which illustrates the use of a recursive function to sum the entries in a chain (see Section 2.12).

The type of the function (and its result) may be specified on the function statement, either before or after the token `recursive`:

```
integer recursive function factorial(n) result(res)
```

or

```
recursive integer function factorial(n) result(res)
```

or in a type declaration statement for the result name (as in Figure 5.13). In fact, the result name, rather than the function name, must be used in any specification statement. In the

[7] This requirement is removed in Fortran 2018, see Section 23.14.

Figure 5.13 Summing the entries in a linked list.

```
recursive function sum(top) result(s)
   type(entry), pointer :: top
   real                 :: s
   if (associated(top)) then
      s = top%value + sum(top%next)
   else
      s = 0.0
   end if
end function sum
```

executable statements, the function name refers to the function itself and the result name must be used for the **result variable**. If there is no `result` clause, the function name is used for the result, and is not available for a recursive function call.

The `result` clause may also be used in a non-recursive function.

Just as in Figure 5.13, any recursive procedure that calls itself directly must contain a conditional test that terminates the sequence of calls at some point, otherwise it will call itself indefinitely.

Each time a recursive procedure is invoked, a fresh set of local data objects is created, which ceases to exist on return. They consist of all data objects declared in the procedure's specification statements or declared implicitly (see Section 8.2), but excepting those with the `data` or `save` attribute (see Sections 8.5 and 8.10) and any dummy arguments. The interface is explicit within the procedure.

Figure 5.14 Library code for one-dimensional integration.

```
   recursive function integrate(f, bounds)
      ! Integrate f(x) from bounds(1) to bounds(2)
      real :: integrate
      interface
         function f(x)
            real             :: f
            real, intent(in) :: x
         end function f
      end interface
      real, dimension(2), intent(in) :: bounds
      ⋮
   end function integrate
```

5.17 Indirect recursion

A procedure may also be invoked by indirect recursion, that is it may call itself through calls to other procedures. To illustrate that this may be useful, suppose we wish to perform a two-dimensional integration but have only the procedure for one-dimensional integration shown in Figure 5.14. For example, suppose that it is desired to integrate a function f of x and y over a rectangle. We might write a Fortran function in a module to receive the value of x as an argument and the value of y from the module itself by host association, as shown in Figure 5.15. We can then integrate over x for a particular value of y, as shown in Figure 5.16, where `integrate` might be as shown in Figure 5.14. We may now integrate over the whole rectangle thus

```
volume = integrate(fy, ybounds)
```

Note that `integrate` calls `fy`, which in turn calls `integrate`.

Figure 5.15 A two-dimensional function to be integrated.

```
module func
   real               :: yval
   real, dimension(2) :: xbounds, ybounds
contains
   function f(xval)
      real             :: f
      real, intent(in) :: xval
      f = ...      ! Expression involving xval and yval
   end function f
end module func
```

Figure 5.16 Integrate over x.

```
function fy(y)
   use func
   real             :: fy
   real, intent(in) :: y
   yval = y
   fy = integrate(f, xbounds)
end function fy
```

5.18 Overloading and generic interfaces

We saw in Section 5.11 how to use a simple interface block to provide an explicit interface to an external or dummy procedure. Another use is for overloading, that is being able to call several procedures by the same **generic identifier**. Here, the interface block contains several interface bodies and the `interface` statement specifies the generic identifier. For example,

the code in Figure 5.17 permits both the functions `sgamma` and `dgamma` to be invoked using the generic name `gamma`.

Figure 5.17 A generic interface block.

```
interface gamma
   function sgamma(x)
      real (selected_real_kind( 6))             :: sgamma
      real (selected_real_kind( 6)), intent(in) :: x
   end function sgamma
   function dgamma(x)
      real (selected_real_kind(12))             :: dgamma
      real (selected_real_kind(12)), intent(in) :: x
   end function dgamma
end interface
```

A specific name for a procedure may be the same as its generic name. For example, the procedure `sgamma` could be renamed `gamma` without invalidating the interface block.

Furthermore, a generic name may be the same as another accessible generic name. In such a case, all the procedures that have this generic name may be invoked through it. This capability is important, since a module may need to extend the intrinsic functions such as `sin` to a new type such as `interval` (Section 3.9).

If it is desired to overload a module procedure, the interface is already explicit so it is inappropriate to specify an interface body. Instead, the statement

[`module` *]* `procedure` *[*`::`*]* *procedure-name-list*

is included in the interface block in order to name the module procedures for overloading; if the functions `sgamma` and `dgamma` were defined in a module, the interface block becomes

```
interface gamma
   module  procedure sgamma, dgamma
end interface
```

It is probably most convenient to place such a block in the module itself.

Any generic specification on an `interface` statement may be repeated on the corresponding `end interface` statement, for example,

```
end interface gamma
```

As for other `end` statements, we recommend use of this fuller form.

Another form of overloading occurs when an interface block specifies a defined operation (Section 3.9) or a defined assignment (Section 3.10) to extend an intrinsic operation or assignment. The scope of the defined operation or assignment is the scoping unit that contains the interface block, but it may be accessed elsewhere by use or host association. If an intrinsic operator is extended, the number of arguments must be consistent with the intrinsic form (for example, it is not possible to define a unary `*` operator).

The general form of the interface block is

```
interface [generic-spec]
   [ interface-body ] ...
   [ [ module ] procedure [ :: ] procedure-name-list ] ...
   ! Interface bodies and module procedure statements
   ! may appear in any order.
end interface [generic-spec]
```

where *generic-spec* is one of

```
generic-name
operator(defined-operator)
assignment(=)
```

A `module procedure` statement is permitted only when a *generic-spec* is present, and all the procedures must be accessible module procedures (as shown in the complete module in Figure 5.21 below). No procedure name may be given a particular *generic-spec* more than once in the interface blocks accessible within a scoping unit. An interface body must be provided for an external or dummy procedure.

If `operator` is specified on the interface statement, all the procedures in the block must be functions with one or two non-optional arguments having intent `in`. If `assignment` is specified, all the procedures must be subroutines with two non-optional arguments, the first having intent `out` or `inout` and the second intent `in`. In order that invocations are always unambiguous, if two procedures have the same generic operator and the same number of arguments or both define assignment, one must have a dummy argument that corresponds by position in the argument list to a dummy argument of the other that has a different type, different kind type parameter, or different rank.

All procedures that have a given generic name must be subroutines or all must be functions, including the intrinsic ones when an intrinsic procedure is extended. Any two non-intrinsic procedures with the same generic name must have arguments that are **distinguishable** in order that any invocation will be unambiguous. Two dummy arguments are distinguishable if

- one is a procedure and the other is a data object,
- they are both data objects or known to be functions but have different types,[8] kind type parameters, or rank,
- one is allocatable and the other is a pointer, or
- one is a function with nonzero rank and the other is not known to be a function.

The rules for any two non-intrinsic procedures with the same generic name are that either

i) one of them has a non-optional data-object dummy argument such that this procedure's number of non-optional data-object dummy arguments that are not distinguishable from it differs from the other procedure's number of such dummy arguments; or

ii) at least one of them has both

[8] Or, if either or both are polymorphic (Section 15.3), they are not type-compatible.

- a non-optional dummy argument that corresponds by position in the argument list to a dummy argument that is distinguishable from it, or for which no dummy argument corresponds by position; and
- a non-optional dummy argument with the same name as a dummy argument that is distinguishable from it, or for which there is no dummy argument of that name.

These two arguments must either be the same or the argument that corresponds by position must occur earlier in the dummy argument list.

For case ii), both rules are needed in order to cater for both keyword and positional dummy argument lists. For instance, the interface in Figure 5.18 is invalid because the two functions are always distinguishable in a positional call, but not on a keyword call such as `f(i=int, x=posn)`. If a generic invocation is ambiguous between a non-intrinsic and an intrinsic procedure, the non-intrinsic procedure is invoked.

Figure 5.18 An example of a broken overloading rule.

```
interface f  ! Invalid interface block
   function fxi(x,i)
      real             :: fxi
      real, intent(in) :: x
      integer          :: i
   end function fxi
   function fix(i,x)
      real             :: fix
      real, intent(in) :: x
      integer          :: i
   end function fix
end interface
```

The interface block in Figure 5.19 illustrates disambiguation based on procedureness: since the compiler always knows whether an actual argument is a procedure, no reference to `g1` could ever be ambiguous.

Figure 5.19 Generic disambiguation based on procedureness.

```
interface g1
  subroutine s1(a)
    real a
  end subroutine
  subroutine s2(a)
    real, external :: a
  end subroutine
end interface
```

The interface block in Figure 5.20 illustrates disambiguation based on whether the argument is a pointer or allocatable: in this case the point of the interface is to allow switching

between using allocatable and pointer, without having to change the name of the deallocation procedure.

Figure 5.20 Generic disambiguation based on `pointer` vs. `allocatable`.

```
interface log_deallocate
  subroutine log_deallocate_real_pointer_2(a)
    real, pointer, intent(inout) :: a(:, :)
  end subroutine
  subroutine log_deallocate_real_allocatable_2(a)
    real, allocatable, intent(inout) :: a(:, :)
  end subroutine
end interface
```

The reason that `allocatable` and `pointer` are only considered to be mutually distinguishable when the pointer does not have intent `in` is that there is an interaction with the automatic targetting feature (see Section 7.18.3) that would have made it possible to write an ambiguous reference.

Note that the presence or absence of the pointer attribute is insufficient to ensure an unambiguous invocation since a pointer actual argument may be associated with a non-pointer dummy argument, see Section 5.7.2.

As already shown, the keyword `module` is optional; for example,

```
interface gamma
   procedure :: sgamma, dgamma
end interface
```

If the keyword `module` is omitted, the named procedures need not be module procedures but may also be external procedures, dummy procedures, or procedure pointers. Each named procedure must already have an explicit interface to be used in this way. This allows, for instance, an external procedure to appear in more than one interface block. For example,

```
type bitstring
   ⋮
end type
⋮
interface operator(*)
   elemental type(bitstring) function bitwise_and(a, b)
      import :: bitstring
      type(bitstring), intent(in) :: a, b
   end function bitwise_and
end interface
interface operator(.and.)
   procedure :: bitwise_and
end interface
```

allows the use of both the `*` and `.and.` operators for 'bitwise and' on values of type `bitstring`.

A generic name is permitted to be the same as a type name and takes precedence over the type name; a structure constructor for the type is interpreted as such only if it cannot be interpreted as a reference to the generic procedure.

There are many scientific applications in which it is useful to keep a check on the sorts of quantities involved in a calculation. For instance, in dimensional analysis, whereas it might be sensible to divide length by time to obtain velocity, it is not sensible to add time to velocity. There is no intrinsic way to do this, but we conclude this section with an outline example, see Figures 5.21 and 5.22, of how it might be achieved using derived types.

Note that definitions for operations between like entities are also required, as shown by `time_plus_time`. Similarly, any intrinsic function that might be required, here `sqrt`, must be overloaded appropriately. Of course, this can be avoided if the components of the variables are referenced directly, as in

```
t%seconds = t%seconds + 1.0
```

5.19 Assumed character length

A character dummy argument may be declared with an asterisk for the value of the length type parameter, in which case it automatically takes the value from the actual argument. For example, a subroutine to sort the elements of a character array might be written thus

```
subroutine sort(n,chars)
   integer, intent(in)                          :: n
   character(len=*), dimension(n), intent(in) :: chars
   ⋮
end subroutine sort
```

If the length of the associated actual argument is needed within the procedure, the intrinsic function `len` (Section 9.7.1) may be invoked, as in Figure 5.23.

An asterisk must not be used for a kind type parameter value. This is because a change of character length is analogous to a change of an array size and can easily be accommodated in the object code, whereas a change of kind probably requires a different machine instruction for every operation involving the dummy argument. A different version of the procedure would need to be generated for each possible kind value of each argument. The overloading feature (previous section) gives the programmer an equivalent functionality with explicit control over which versions are generated.

5.20 The subroutine and function statements

We finish this chapter by giving the syntax of the `subroutine` and `function` statements, which have so far been explained through examples. It is

[prefix] `subroutine` *subroutine-name [* `(` *[dummy-argument-list]* `)` *]*

and

[prefix] `function` *function-name* `(` *[dummy-argument-list]* `)` *[* `result(`*result-name*`)` *]*

Figure 5.21 A module for distinguishing real entities.

```
module sorts
   type time
      real :: seconds
   end type time
   type velocity
      real :: metres_per_second
   end type velocity
   type length
      real :: metres
   end type length
   type length_squared
      real :: metres_squared
   end type length_squared
   interface operator(/)
      module procedure length_by_time
   end interface
   interface operator(+)
      module procedure time_plus_time
   end interface
   interface sqrt
      module procedure sqrt_metres_squared
   end interface
contains
   function length_by_time(s, t)
      type(length), intent(in) :: s
      type(time), intent(in)   :: t
      type(velocity)           :: length_by_time
      length_by_time%metres_per_second = s%metres / t%seconds
   end function length_by_time
   function time_plus_time(t1, t2)
      type(time), intent(in)   :: t1, t2
      type(time)               :: time_plus_time
      time_plus_time%seconds = t1%seconds + t2%seconds
   end function time_plus_time
   function sqrt_metres_squared(l2)
      type(length_squared), intent(in) :: l2
      type(length)                     :: sqrt_metres_squared
      sqrt_metres_squared%metres = sqrt(l2%metres_squared)
   end function sqrt_metres_squared
end module sorts
```

Figure 5.22 Use of the module of Figure 5.21.

```
program test
   use sorts
   type(length)         :: s  = length(10.0), l
   type(length_squared) :: s2 = length_squared(10.0)
   type(velocity)       :: v
   type(time)           :: t  = time(3.0)
   v = s / t
   ! Note: v = s + t   or   v = s * t  would be invalid.
   t = t + time(1.0)
   l = sqrt(s2)
   print *, v, t, l
end program test
```

Figure 5.23 A function with an argument of assumed character length.

```
   ! Count the number of occurrences of letter in string.
   integer function count (letter, string)
      character (1), intent(in) :: letter
      character (*), intent(in) :: string
      count = 0
      do i = 1, len(string)
         if (string(i:i) == letter) count = count + 1
      end do
   end function count
```

where *prefix* is

prefix-spec [prefix-spec] ...

and *prefix-spec* is *type*, `recursive`, `pure`, or `elemental`. A *prefix-spec* must not be repeated. For details of *type*, see Section 8.17; this, of course, must not be present on a `subroutine` statement.

Apart from `pure` and `elemental`, which will be explained in Sections 7.8 and 7.9, each feature has been explained separately and the meanings are the same in the combinations allowed by the syntax.

5.21 Summary

A program consists of a sequence of program units. It must contain exactly one main program but may contain any number of modules and external subprograms. We have described each kind of program unit. Modules contain data definitions, derived-type definitions, namelist groups, interface blocks and the import statement, and module subprograms, all of which may be accessed in other program units with the `use` statement. The program units may be in any order, but many compilers require modules to precede their use.

Subprograms define procedures, which may be functions or subroutines. They may also be defined intrinsically (Chapter 9), and external procedures may be defined by means other than Fortran. We have explained how information is passed between program units and to procedures through argument lists and through the use of modules. Procedures may be called recursively provided they are correspondingly specified.

The interface to a procedure may be explicit or implicit. If it is explicit, keyword calls may be made, and the procedure may have optional arguments. Interface blocks permit procedures to be invoked as operations or assignments, or by a generic name. The character lengths of dummy arguments may be assumed.

We have also explained about the scope of labels and Fortran names, and introduced the concept of a scoping unit.

Exercises

1. A subroutine receives as arguments an array of values, x, and the number of elements in x, n. If the mean and variance of the values in x are estimated by

$$\text{mean} = \frac{1}{n}\sum_{i=1}^{n} x(i)$$

and

$$\text{variance} = \frac{1}{n-1}\sum_{i=1}^{n} (x(i) - \text{mean})^2$$

write a subroutine which returns these calculated values as arguments. The subroutine should check for invalid values of n (≤ 1).

2. A subroutine `matrix_mult` multiplies together two matrices A and B, whose dimensions are $i \times j$ and $j \times k$, respectively, returning the result in a matrix C dimensioned $i \times k$. Write `matrix_mult`, given that each element of C is defined by

$$C(m,n) = \sum_{\ell=1}^{j} (A(m,\ell) \times B(\ell,n))$$

The matrices should appear as arguments to `matrix_mult`.

3. The subroutine `random_number` (Section 9.18.3) returns a random number in the range 0.0 to 1.0, that is

```
call random_number(r)    ! 0≤r<1
```

Using this function, write the subroutine `shuffle` of Figure 5.4.

4. A character string consists of a sequence of letters. Write a function to return that letter of the string which occurs earliest in the alphabet; for example, the result of applying the function to `DGUMVETLOIC` is `C`.

5. Write an internal procedure to calculate the volume, $\pi r^2 \ell$, of a cylinder of radius r and length ℓ, using as the value of π the result of `acos(-1.0)`, and reference it in a host procedure.

6. For a simple card game of your own choice, and using the random number procedure (Section 9.18.3), write the subroutines `deal` and `play` of Section 5.4, using data in a module to communicate between them.

7. Objects of the intrinsic type `character` are of a fixed length. Write a module containing a definition of a variable-length character string type, of maximum length 80, and also the procedures necessary to:

i) assign a character variable to a string;

ii) assign a string to a character variable;

iii) return the length of a string;

iv) concatenate two strings.

6. Allocation of data

6.1 Introduction

There is an underlying assumption in Fortran that the processor supplies a mechanism for managing heap storage. (A heap is a memory management mechanism whereby fresh storage may be established and old storage may be discarded in any order. Mechanisms to deal with the progressive fragmentation of the memory are usually required.) The statements described in this chapter are the user interface to that mechanism.

6.2 The allocatable attribute

As we have seen in Section 2.10.2, sometimes an array is required only after some data have been read or some calculations performed. For this purpose, an object may be given the `allocatable` attribute by a statement such as

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

Its rank is specified when it is declared, but the bounds (if it is an array) are undefined until an `allocate` statement has been executed for it. Its initial status is unallocated and it becomes allocated following successful execution of an `allocate` statement.

An important example is shown in Figure 6.1. The array `work` is placed in a module and is allocated at the beginning of the main program to a size that depends on input data. The array is then available throughout program execution in any subprogram that has a `use` statement for `work_array`.

Figure 6.1 An allocatable array in a module.

```
module work_array
   integer                                :: n
   real, dimension(:,:,:), allocatable :: work
end module work_array
program main
   use work_array
   read *, n
   allocate (work(n, 2*n, 3*n))
   ⋮
```

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

When an allocatable object `a` is no longer needed, it may be deallocated by execution of the statement

```
deallocate (a)
```

following which the object is unallocated. The `deallocate` statement is described in more detail in Section 6.6.

If it is required to make any change to the bounds of an allocatable array, the array must be deallocated and then allocated afresh. This can be carried out either by explicit deallocation and reallocation, or automatically (see Section 6.7). It is an error to allocate an allocatable array that is already allocated, or to deallocate an allocatable array that is unallocated, but one that can easily be avoided by the use of the `allocated` intrinsic function (Section 9.2) to enquire about the allocation status.

An undefined allocation status cannot occur. On return from a subprogram, an allocated allocatable object without the `save` attribute (Section 8.10) is automatically deallocated if it is local to the subprogram.

6.3 Deferred type parameters

A `len` type parameter value is permitted to be a colon in a type declaration statement such as

```
character(len=:), pointer :: varchar
```

for a pointer or an allocatable entity. It indicates a **deferred type parameter**; such a type parameter has no defined value until it is given one by allocation or pointer assignment. For example, in

```
character(:), pointer :: varchar
character(100), target :: name
character(200), target :: address
⋮
varchar => name
⋮
varchar => address
```

the character length of `varchar` after each pointer assignment is the same as that of its target; that is, 100 after the first pointer assignment and 200 after the second.

For intrinsic types, only character length may be deferred. Derived types that are parameterized may have type parameters which can be deferred, see Section 13.2.2.

Deferred type parameters can be given values by the `allocate` statement, see Section 6.5. For allocatable variables, they can also be given values by assignment, see Section 6.7.

6.4 Allocatable scalars

The `allocatable` attribute (and hence the `allocated` function) may also be applied to scalar variables and components. This is particularly useful when combined with deferred type parameters, for example, in

```
character(:), allocatable :: chdata
integer                   :: unit, reclen
⋮
read *, reclen
allocate (character(reclen) :: chdata)
read *, chdata
```

where `reclen` allows the length of `character` to be specified at run time.

6.5 The allocate statement

The general form of the `allocate` statement is

`allocate (` *[type-spec* `::` *] allocation-list [*, *alloc-spec]* ... `)`

where *allocation-list* is a list of allocations of the form

allocate-object [`(` *array-bounds-list* `)` *]*

each *array-bound* has the form

[lower-bound `:` *] upper-bound*

and *alloc-spec* is one of

`errmsg=`***erm***
`mold=`***expr***
`source=`***expr***
`stat=`***stat***

where no specifier may appear more than once, *stat* is a scalar integer variable, and *erm* is a scalar default character variable. Neither *stat* nor *erm* may be part of an object being allocated.

The optional *type-spec*, and `mold=` and `source=` specifiers, are discussed in Section 15.4.

If the `stat=` specifier is present, *stat* is given either the value zero after a successful allocation or a positive value after an unsuccessful allocation (for example, if insufficient storage is available). After an unsuccessful execution, each array that was not successfully allocated retains its previous allocation or pointer association status. If `stat=` is absent and the allocation is unsuccessful, execution stops.

If the `errmsg=` specifier is present and an error during allocation occurs, an explanatory message is assigned to the variable. For example,

```
character(200) :: error_message ! Probably long enough
⋮
allocate (x(n), stat=allocate_status, errmsg=error_message)
if (allocate_status > 0) then
   print *, 'Allocation of X failed:', trim(error_message)
   ⋮                                  ! trim is described in Chapter 9
end if
```

This is helpful, because the error codes available in the *stat* variable are processor dependent.

Each *allocate-object* is allocatable or a pointer. It is permitted to have zero character length.

Each *lower-bound* and each *upper-bound* is a scalar integer expression. The default value for the lower bound is 1. The number of *array-bound*s in a list must equal the rank of the *allocate-object*. They determine the array bounds, which do not alter if the value of a variable in one of the expressions changes subsequently. An array may be allocated to be of size zero.

The bounds of all the arrays being allocated are regarded as undefined during the execution of the `allocate` statement, so none of the expressions that specify the bounds may depend on any of the bounds or on the value of the `stat=` variable. For example,

```
allocate (a(size(b)), b(size(a)))   ! invalid
```

or even

```
allocate (a(n), b(size(a)))         ! invalid
```

is not permitted, but

```
allocate (a(n))
allocate (b(size(a)))
```

is valid. This restriction allows the processor to perform the allocations in a single `allocate` statement in any order.

In contrast to the case with an allocatable object, a pointer may be allocated a new target even if it is currently associated with a target. In this case the previous association is broken. If the previous target was created by allocation, it becomes inaccessible unless another pointer is associated with it. Linked lists are normally created by using a single pointer in an `allocate` statement for each node of the list. There is an example in Figure 4.9.

6.6 The deallocate statement

When an allocatable object or pointer target is no longer needed, its storage may be recovered by using the `deallocate` statement. Its general form is

`deallocate (` *allocate-object-list* `[,stat=`*stat*`] [, errmsg=`*erm* `]`)

where each *allocate-object* is an allocatable object that is allocated or a pointer that is associated with the whole of a target that was allocated through a pointer in an `allocate` statement.[1] A pointer becomes disassociated (Section 3.3) following successful execution of the statement. Here, *stat* is a scalar integer variable that must not be deallocated by the statement nor depend on an object that is deallocated by the statement. If `stat=` is present, *stat* is given either the value zero after a successful execution or a positive value after an unsuccessful execution (for example, if a pointer is disassociated). If an error during deallocation occurs, an explanatory message is assigned to the optional `errmsg=` variable. After an unsuccessful execution, each object that was not successfully deallocated retains its previous allocation or pointer association status. If `stat=` is absent and the deallocation is unsuccessful, execution stops.

If there is more than one object in the list, there must be no dependencies among them, to allow the processor to deallocate the objects one by one in any order. An object must not

[1] Note that this excludes a pointer that is associated with an allocatable array.

be deallocated while it or a subobject of it is associated with a dummy argument; this can happen, for example, if the object is in a module.

A danger in using the `deallocate` statement is that an object may be deallocated while pointers are still associated with it. Such pointers are left 'dangling' in an undefined state, and must not be reused until they are again associated with an actual target.

In order to avoid an accumulation of unused and unusable storage, all explicitly allocated storage should be explicitly deallocated when it is no longer required (this is automatic for unsaved local allocatable variables, see the end of Section 6.2). This explicit management is required in order to avoid a potentially significant overhead on the part of the processor in handling arbitrarily complex allocation and reference patterns.

Note also that the standard does not specify whether the processor recovers storage that held a target allocated through a pointer but no longer accessible through this or any other pointer. This failure to recover storage is known as **memory leakage**. It might be important where, for example, a pointer function is referenced within an expression – the programmer cannot rely on the compiler to arrange for deallocation. To ensure that there is no memory leakage, it is necessary to use such functions only on the right-hand side of pointer assignments or as pointer component values in structure constructors, and to deallocate the pointer when it is no longer needed.

Allocatable objects have the advantage that they do not leak memory, because an allocatable object cannot be allocated if already allocated, and is automatically deallocated on return from a subprogram if it is local to the subprogram and does not have the `save` attribute.

6.7 Automatic reallocation

Automatic reallocation of arrays is possible, and simplifies the use of array functions which return a variable-sized result (such as the intrinsic functions `pack` and `unpack`).

For example, in

```
subroutine process(x)
   real(wp), intent(inout) :: x(:)
   real(wp), allocatable   :: nonzero_values(:)
   nonzero_values = pack(x, x/=0)
```

the array `nonzero_values` is automatically allocated to be of the correct length to contain the results of the intrinsic function `pack`, instead of the user having to allocate it manually (which would necessitate counting the number of nonzeros separately). It also permits a simple extension of an existing allocatable array whose lower bounds are all 1. To add some extra values to such an integer array `a` of rank 1, it is sufficient to write, for example,

```
a = [ a, 5, 6 ]
```

This automatic reallocation also occurs if the allocatable variable has a deferred type parameter which does not already have the same value as the corresponding parameter of the expression. This applies to allocatable scalars as well as to allocatable arrays, as in

```
character(:), allocatable :: quotation
⋮
quotation = 'Now is the winter of our discontent.'
⋮
quotation = "This ain't the summer of love."
```

In each of the assignments to `quotation`, it is reallocated to be the right length (unless it is already of that length) to hold the desired quotation. If instead the normal truncation or padding is required in an assignment to an allocatable-length character, substring notation can be used to suppress the automatic reallocation. For example,

```
quotation(:) = ''
```

leaves `quotation` at its current length, setting all of it to blanks.

Automatic reallocation only occurs for normal intrinsic assignment, and not for defined assignment or for `where` constructs (Section 7.6).

6.8 Transferring an allocation

The intrinsic subroutine `move_alloc` allows an allocation to be moved from one allocatable object to another. We defer the coarray case to Section 17.6. Without coarrays, the subroutine is pure (Section 7.8).

`call move_alloc (from, to)` where:

`from` is allocatable and of any type. It has intent `inout`. It must not be coindexed (Section 17.4).

`to` is allocatable and of the same type and rank as `from`. It has intent `out`. It must not be coindexed (Section 17.4).

After the call, the allocation status of `to` is that of `from` beforehand and `from` becomes deallocated. If `to` has the `target` attribute, any pointer that was associated with `from` will be associated with `to`; otherwise, such a pointer will become undefined.

It provides what is essentially the allocatable equivalent of pointer assignment: allocation transfer. However, unlike pointer assignment, this maintains the allocatable semantics of having at most one allocated object for each allocatable variable. For example,

```
real, allocatable :: a1(:), a2(:)
allocate (a1(0:10))
a1(3) = 37
call move_alloc(from=a1, to=a2)
! a1 is now unallocated,
! a2 is allocated with bounds (0:10) and a2(3)==37.
```

The subroutine `move_alloc` can be used to minimize the amount of copying required when one wishes to expand or contract an allocatable array; the canonical sequence for this is:

```
real, allocatable :: a(:,:), temp(:,:)
⋮
! Increase size of a to (n, m)
allocate (temp(n, m))
temp(1:size(a,1), 1:size(a,2)) = a
call move_alloc(temp, a)
! a now has shape (/ n, m /), and temp is unallocated
```

This sequence only requires one copying operation instead of the two that would have been required without `move_alloc`. Because the copy is controlled by the user, pre-existing values will end up where the user wants them (which might be at the same subscripts, or all at the beginning, or all at the end, etc.).

6.9 Allocatable dummy arguments

A dummy argument is permitted to have the allocatable attribute. In this case, the corresponding actual argument must be allocatable and of the same type, kind parameters, and rank; also, the interface must be explicit. The dummy argument always receives the allocation status (descriptor) of the actual argument on entry and the actual argument receives that of the dummy argument on return. In both cases, this may be unallocated. If allocated, the bounds are also received (as for an array pointer but not an assumed-shape array).

Our expectation is that some compilers will perform copy-in copy-out of the descriptor. Rule i) of Section 5.7.3 is applicable and is designed to permit compilers to do this. In particular, this means that no reference to the actual argument (for example, through it being a module variable) is permitted from the invoked procedure if the dummy array is allocated or deallocated there.

For the object itself, the situation is just like the case when the actual and dummy arguments are both explicit-shape arrays (see Section 5.7.4). Copy-in copy-out is permitted unless both objects have the `target` attribute.

An allocatable dummy argument is permitted to have intent and this applies both to the allocation status (the descriptor) and to the object itself. If the intent is `in`, the object is not permitted to be allocated or deallocated and the value is not permitted to be altered. If the intent is `out` and the object is allocated on entry, it becomes deallocated. An example of the application of an allocatable dummy argument to reading arrays of variable bounds is shown in Figure 6.2.

6.10 Allocatable functions

A function result is permitted to have the allocatable attribute, which is very useful when the size of the result depends on a calculation in the function itself, as illustrated in Figure 6.3. The allocation status on each entry to the function is unallocated. The result may be allocated and deallocated any number of times during execution of the procedure, but it must be allocated and have a defined value on return.

Figure 6.2 Reading arrays whose size is not known beforehand.

```
subroutine load(array, unit)
   real, allocatable, intent(out), dimension(:, :, :) :: array
   integer, intent(in)          :: unit
   integer                      :: n1, n2, n3
   read *, n1, n2, n3
   allocate (array(n1, n2, n3))
   read *, array
end subroutine load
```

The interface must be explicit in any scoping unit in which the function is referenced. The result is automatically deallocated after execution of the statement in which the reference occurs, even if it has the `target` attribute.

Figure 6.3 An allocatable function to remove duplicate values.

```
program no_leak
   real, dimension(100) :: x, y
   ⋮
   y(:size(compact(x))) = compact(x)**2
   ⋮
contains
   function compact(x) ! To remove duplicates from the array x
      real, allocatable, dimension(:) :: compact
      real, dimension(:), intent(in)  :: x
      integer                         :: n
      ⋮                ! Find the number of distinct values, n
      allocate (compact(n))
      ⋮                ! Copy the distinct values into compact
   end function compact
end program no_leak
```

6.11 Allocatable components

Components of a derived type are permitted to have the `allocatable` attribute. For example, a lower-triangular matrix may be held by using an allocatable array for each row. Consider the type

```
type row
   real, dimension(:), allocatable :: r
end type row
```

and the arrays

```
type(row), dimension(n) :: s, t     ! n of type integer
```

Storage for the rows can be allocated thus

```
do i = 1, n               ! i of type integer
   allocate (t(i)%r(1:i)) ! Allocate row i of length i
end do
```

The array assignment

```
s = t
```

would then be equivalent to the assignments

```
s(i)%r = t(i)%r
```

for all the components.

For an object of a derived type that has a component of derived type, we need the concept of an **ultimate allocatable component**, which is an ultimate component (Section 2.13) that is allocatable. Just as for an ordinary allocatable object, the initial state of an ultimate allocable component is unallocated. Hence, there is no need for default initialization of allocatable components. In fact, initialization in a derived-type definition of an allocatable component is not permitted, see Section 8.5.5.

In a structure constructor (Section 3.9), an expression corresponding to an allocatable component must be an object or a reference to the intrinsic function `null` with no arguments. If it is an allocatable object, the component takes the same allocation status and, if allocated, the same bounds and value. If it is an object, but not allocatable, the component is allocated with the same bounds and is assigned the same value. If it is a reference to the intrinsic function `null` with no arguments, the component receives the allocation status of unallocated.

Allocatable components are illustrated in Figure 6.4, where code to manipulate polynomials with variable numbers of terms is shown.

An object of a type having an ultimate allocatable component is permitted to have the `parameter` attribute (be a constant). In this case the component is always unallocated. It is not permitted to appear in an `allocate` statement.

When a variable of derived type is deallocated, any ultimate allocatable component that is allocated is also deallocated, as if by a `deallocate` statement. The variable may be a pointer or allocatable, and the rule applies recursively, so that all allocated allocatable components at all levels (apart from any lying beyond pointer components) are deallocated. Such deallocations of components also occur when a variable is associated with an intent `out` dummy argument.

Intrinsic assignment

variable = *expr*

for a type with an ultimate allocatable component (as in `r = p + q` in Figure 6.4) consists of the following steps for each such component.

i) If the component of *variable* is allocated, it is deallocated.

ii) If the component of *expr* is allocated, the component of *variable* is allocated with the same bounds and the value is then transferred using intrinsic assignment.

Figure 6.4 Using allocatable components for adding polynomials.

```
module real_polynomial_module
   type real_polynomial
     real, allocatable, dimension(:) :: coeff
   end type real_polynomial
   interface operator(+)
      module procedure rp_add_rp
   end interface operator(+)
contains
   function rp_add_rp(p1, p2)
      type(real_polynomial)              :: rp_add_rp
      type(real_polynomial), intent(in) :: p1, p2
      integer                            :: m, m1, m2
      m1 = ubound(p1%coeff,1)
      m2 = ubound(p2%coeff,1)
      allocate (rp_add_rp%coeff(max(m1,m2)))
      m = min(m1,m2)
      rp_add_rp%coeff(:m) = p1%coeff(:m) +p2%coeff(:m)
      if (m1 > m) rp_add_rp%coeff(m+1:) = p1%coeff(m+1:)
      if (m2 > m) rp_add_rp%coeff(m+1:) = p2%coeff(m+1:)
    end function rp_add_rp
end module real_polynomial_module
program example
   use real_polynomial_module
   type(real_polynomial) :: p, q, r
   p = real_polynomial((/4.0, 2.0, 1.0/))  ! Set p to 4+2x+x**2
   q = real_polynomial((/-1.0, 1.0/))
   r = p + q
   print *, 'Coefficients are: ', r%coeff
end program example
```

If the allocatable component of *expr* is unallocated, nothing happens in step ii), so the component of *variable* is left unallocated. Note that if the component of *variable* is already allocated with the same shape, the compiler may choose to avoid the overheads of deallocation and reallocation. Note also that if the compiler can tell that there will be no subsequent reference to *expr*, because it is a function reference or a temporary variable holding the result of expression evaluation, no allocation or assignment is needed – all that has to happen is the deallocation of any allocated ultimate allocatable components of *variable* followed by the copying of the descriptor.

If a component is itself of a derived type with an allocatable component, the intrinsic assignment in step ii) will involve these rules, too. In fact, they are applied recursively at all levels, and copying occurs in every case. This is known as **deep copying**, as opposed to

shallow copying, which occurs for pointer components, where the descriptor is copied and nothing is done for components of pointer components.

If an actual argument and the corresponding dummy argument have an ultimate allocatable component, rule i) of Section 5.7.3 is applicable and requires all allocations and deallocations of the component to be performed through the dummy argument, in case copy-in copy-out is in effect.

If a statement contains a reference to a function whose result is of a type with an ultimate allocatable component, any allocated ultimate allocatable components of the function result are deallocated after execution of the statement. This parallels the rule for allocatable function results (Section 6.10).

Figure 6.5 Allocatable list example. The `read` statement here is explained in Section 10.7.

```
type my_real_list
  real value
  type(my_real_list), allocatable :: next
end type
type(my_real_list), allocatable, target :: list
type(my_real_list), pointer :: last
real :: x
⋮
last => null()
do
  read (unit, *, iostat=ios) x
  if (ios/=0) exit
  if (.not.associated(last)) then
    allocate (list, source=my_real_list(x))
    last => list
  else
    allocate (last%next, source=my_real_list(x))
    last => last%next
  end if
end do
! list now contains all the input values, in order of reading.
⋮
deallocate (list) ! deallocates every element in the list.
```

6.11.1 Allocatable components of recursive type

An allocatable component is permitted to be of any derived type, including the type being defined or a type defined later in the program unit. This can be used to define dynamic structures without involving pointers, thus gaining the usual benefits of allocatable variables: no aliasing (except where the `target` attribute is used), contiguity, and automatic

deallocation. Automatic deallocation means that deallocating the parent variable (or returning from the procedure in which it is defined) will completely deallocate the entire dynamic structure. Figure 6.5 shows how this can be used to build a list (the `source=` clause causes the type, type parameters, and values to be copied and is explained later in Section 15.4). In building up the list in that example, it was convenient to use a pointer to the end of the list. If, on the other hand, we want to insert a new value somewhere else (such as at the beginning of the list), careful use of the `move_alloc` intrinsic (Section 6.8) is recommended to avoid making temporary copies of the entire list. We illustrate this with the subroutine `push` for adding an element to the top of a stack in Figure 6.6. Similar comments apply to element deletion, illustrated by subroutine `pop` in Figure 6.6.

Figure 6.6 Allocatable stack procedures.

```
subroutine push(list, newvalue)
  type(my_real_list), allocatable :: list, temp
  real, intent(in)                :: newvalue
  call move_alloc(list, temp)
  allocate (list, source=my_real_list(x))
  call move_alloc(temp, list%next)
end subroutine
subroutine pop(list)
  type(my_real_list), allocatable :: list, temp
  call move_alloc(list%next, temp)
  call move_alloc(temp, list)
end subroutine
```

One might imagine that the compiler would produce similar code (that is, code avoiding deep copies) for the much simpler statements

```
list = my_real_list(newvalue, list)
```

and

```
list = list%next
```

as the executable parts of `push` and `pop`, respectively, but in fact the model for allocatable assignment in the standard specifies automatic deallocation only when an array shape, length type parameter, or dynamic type differs; that is not the case in these examples, so the compiler is expected to perform deep copying. (A standard-conforming program can only tell the difference when the type has any final subroutines or the list has the `target` attribute; so if the variables involved are not polymorphic and not targets, a compiler might produce more optimal code.)

6.12 Allocatable arrays vs. pointers

Why are allocatable arrays needed? Is all their functionality not available (and more) with pointers? The reason is that there are significant advantages for memory management and

execution speed in using allocatable arrays when the added functionality of pointers is not needed.

- Code for an array pointer is likely to be less efficient because allowance has to be made for strides other than unity. For example, its target might be the section `vector(1:n:2)` or the section `matrix(i,1:n)` with non-unit strides, whereas most computers hold allocatable arrays in contiguous memory.

- If a defined operation involves a temporary variable of a derived type with a pointer component, the compiler will probably be unable to deallocate its target when storage for the variable is freed. Consider, for example, the statement

  ```
  a = b + c*d      ! a, b, c, and d are of the same derived type
  ```

  This will create a temporary for `c*d`, which is not needed once `b + c*d` has been calculated. The compiler is unlikely to be sure that no other pointer has the component or part of it as a target, so is unlikely to deallocate it.

- Intrinsic assignment is often unsuitable for a derived type with a pointer component because the assignment

  ```
  a = b
  ```

  will leave `a` and `b` sharing the same target for their pointer component. Therefore, a defined assignment that allocates a fresh target and copies the data will be used instead. However, this is very wasteful if the right-hand side is a temporary such as that of the assignment of the previous paragraph.

- Similar considerations apply to a function invocation within an expression. The compiler will be unlikely to be able to deallocate the pointer after the expression has been calculated.

- When a variable of derived type is deallocated, any ultimate allocatable component that is allocated is also deallocated. To avoid memory leakage with pointer components, the programmer would need to deallocate each one explicitly and be careful to order the deallocations correctly.

Although the Fortran standard does not mention descriptors, it is very helpful to think of an allocatable array as being held as a descriptor that records whether it is allocated and, if so, its address and its bounds in each dimension. This is like a descriptor for a pointer, but no strides need be held since these are always unity. As for pointers, the expectation is that the array itself is held separately.

6.13 Summary

We have described how storage allocation for an object may be controlled in detail by a program.

Exercises

1. Using the type `real_polynomial` of Figure 6.4 in Section 6.11, write code to define a variable of that type with an allocatable component length of four and then to extend that allocatable array with two additional values.

2. Given the type

```
type emfield
   real, allocatable :: strength(:,:)
end type
```

initialize a variable of type `emfield` so that its component has bounds (1:4,1:6) and value 1 everywhere. Extend this variable so that the component has bounds (0:5,0:8), keeping the values of the old elements and setting the values of the new elements to zero.

3. As Exercise 2, but with new bounds (1:6,1:9) and using the `reshape` intrinsic function.

4. Write a statement that makes an existing rank-2 integer array `b`, that has lower bounds of 1, two rows and two columns larger, with the old elements' values retained in the middle of the array. (Hint: Use the `reshape` intrinsic function.)

7. Array features

7.1 Introduction

In an era when many computers have the hardware capability for efficient processing of array operands, it is self-evident that a numerically based language such as Fortran should have matching notational facilities. Such facilities provide not only notational convenience for the programmer, but also provide an opportunity to enhance optimization.

Arrays were introduced in Sections 2.10 to 2.12, their use in simple expressions and in assignments was explained in Sections 3.11 and 3.12, and they were used as procedure arguments in Chapter 5. These descriptions were deliberately restricted because Fortran contains a very full set of array features whose complete description would have unbalanced those chapters. The purpose of this chapter is to describe the array features in detail, but without anticipating the descriptions of the array intrinsic procedures of Chapter 9; the rich set of intrinsic procedures should be regarded as an integral part of the array features.

7.2 Zero-sized arrays

It might be thought that an array would always have at least one element. However, such a requirement would force programs to contain extra code to deal with certain natural situations. For example, the code in Figure 7.1 solves a lower-triangular set of linear equations. When `i` has the value `n` the sections have size zero, which is just what is required.

Figure 7.1 A do loop whose final iteration has a zero-sized array.

```
do i = 1,n
   x(i) = b(i) / a(i, i)
   b(i+1:n) = b(i+1:n) - a(i+1:n, i) * x(i)
end do
```

Fortran allows arrays to have zero size in all contexts. Whenever a lower bound exceeds the corresponding upper bound, the array has size zero.

There are few special rules for zero-sized arrays because they follow the usual rules, though some care may be needed in their interpretation. For example, two zero-sized arrays of the same rank may have different shapes. One might have shape (0,2) and the other (0,3) or (2,0).

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

Such arrays of differing shape are not conformable and therefore may not be used together as the operands of a binary operation. However, an array is always conformable with a scalar so the statement

zero-sized-array = *scalar*

is valid and the scalar is 'broadcast to all the array elements', making this a 'do nothing' statement.

A zero-sized array is regarded as being defined always, because it has no values that can be undefined.

7.3 Automatic objects

A procedure with dummy arguments that are arrays whose size varies from call to call may also need local arrays whose size varies. A simple example is the array `work` in the subroutine to interchange two arrays that is shown in Figure 7.2.

Figure 7.2 A procedure with an automatic array; `size` is described in Section 9.14.2.

```
subroutine swap(a, b)
   real, dimension(:), intent(inout) :: a, b
   real, dimension(size(a))          :: work ! automatic array
            ! size provides the size of an array
   work = a
   a = b
   b = work
end subroutine swap
```

An array whose extents vary in this way is called an **automatic array**, and is an example of an **automatic data object**. Such an object is not a dummy argument and its declaration contains one or more values that are not known at compile time; that is, not a constant expression (Section 8.4). An implementation is likely to bring them into existence when the procedure is called and destroy them on return, maintaining them on a stack.[1] The values must be defined by specification expressions (Section 8.18).

Another way that automatic objects arise is through varying character length. The variable `word2` in

```
subroutine example(word1)
   character(len = *), intent(inout) :: word1
   character(len = len(word1))       :: word2
```

is an example. If a function result has varying character length, the interface must be explicit at the point of call because the compiler needs to know this, as shown in Figure 7.3.

A parameterized derived type (Section 13.2) may have a length type parameter that behaves very like character length, and a dummy argument of such a type may be an automatic object. We defer discussion of this to Section 13.2.2.

[1] A stack is a memory management mechanism whereby fresh storage is established and old storage is discarded on a 'last in, first out' basis, often within contiguous memory.

Figure 7.3 A module containing a procedure with an automatic scalar.

```
program loren
   character (len = *), parameter :: a = 'just a simple test'
   print *, double(a)
contains
   function double(a)
      character (len = *), intent(in) :: a
      character (len = 2*len(a))      :: double
      double = a//a
   end function double
end program loren
```

An array bound or the character length of an automatic object is fixed for the duration of each execution of the procedure and does not vary if the value of the specification expression varies or becomes undefined.

An automatic object must not be given the `save` attribute, see Section 8.10, because this is obviously contrary to being automatic; and it must not be given an initial value, see Sections 8.5.1 and 8.5.2, because this gives it the `save` attribute.

7.4 Elemental operations and assignments

We saw in Section 3.11 that an intrinsic operator can be applied to conformable operands, to produce an array result whose element values are the values of the operation applied to the corresponding elements of the operands. Such an operation is called **elemental**.

It is not essential to use operator notation to obtain this effect. Many of the intrinsic procedures (Chapter 9) are elemental and have scalar dummy arguments that may be called with array actual arguments provided all the array arguments have the same shape. Such a reference is called an **elemental reference**. For a function, the shape of the result is the shape of the array arguments. For example, we may find the square roots of all the elements of a real array `a` thus:

```
a = sqrt(a)
```

If any actual argument in a subroutine invocation is array valued, all the actual arguments corresponding to dummy arguments with intent `out` or `inout` must be arrays. If a procedure that invokes an elemental procedure has an optional array-valued dummy argument that is absent, that dummy argument must not be used as an actual argument in the elemental invocation unless another array of the same rank is associated with a non-optional argument of the elemental procedure (to ensure that the rank does not vary from call to call).

Similarly, an intrinsic assignment may be used to assign a scalar to all the elements of an array, or to assign each element of an array to the corresponding element of an array of the same shape (Section 3.12). Such an assignment is also called **elemental**.

For a defined operator, a similar effect may be obtained with a generic interface to functions for each desired rank or pair of ranks. For example, the module in Figure 7.4 provides

Figure 7.4 Interval addition for scalars and arrays of rank one.

```
module interval_addition
   type interval
      real :: lower, upper
   end type interval
   interface operator(+)
      module procedure add00, add11
   end interface
contains
   function add00 (a, b)
      type (interval)             :: add00
      type (interval), intent(in) :: a, b
      add00%lower = a%lower + b%lower  ! Production code would
      add00%upper = a%upper + b%upper  ! allow for roundoff.
   end function add00
   function add11 (a, b)
      type (interval), dimension(:), intent(in)       :: a
      type (interval), dimension(size(a))             :: add11
      type (interval), dimension(size(a)), intent(in) :: b
      add11%lower = a%lower + b%lower  ! Production code would
      add11%upper = a%upper + b%upper  ! allow for roundoff.
   end function add11
end module interval_addition
```

summation for scalars and rank-one arrays of intervals (Section 3.9). Alternatively, an elemental procedure can be defined for this purpose (Section 7.9).

Similarly, elemental versions of defined assignments may be provided explicitly or an elemental procedure can be defined for this purpose (Section 7.9).

7.5 Array-valued functions

We mentioned in Section 5.10 that a function may have an array-valued result, and have used this language feature in Figure 7.4 where the interpretation is obvious.

In order that the compiler should know the shape of the result, the interface must be explicit (Section 5.11) whenever such a function is referenced. The shape is specified within the function definition by the `dimension` attribute for the function name. Unless the function result is allocatable or a pointer, the bounds must be explicit expressions and they are evaluated on entry to the function. For another example, see the declaration of the function result in Figure 7.5.

An array-valued function is not necessarily elemental. For example, at the end of Section 3.11 we considered the type

Figure 7.5 A function for matrix by vector multiplication; `size` is defined in Section 9.14.

```
function mult(a, b)
   type(matrix), dimension(:, :)       :: a
   type(matrix), dimension(size(a, 2)) :: b
   type(matrix), dimension(size(a, 1)) :: mult
   integer                             :: j, n

   mult = 0.0     ! A defined assignment from a real
                  ! scalar to a rank-one matrix.
   n = size(a, 1)
   do j = 1, size(a, 2)
      mult = mult + a(1:n, j) * b(j)
            ! Uses defined operations for addition of
            ! two rank-one matrices and multiplication
            ! of a rank-one matrix by a scalar matrix.
   end do
end function mult
```

```
type matrix
   real :: element
end type matrix
```

Its scalar and rank-one operations might be as for reals, but for multiplying a rank-two array by a rank-one array, we might use the module function shown in Figure 7.5 to provide matrix by vector multiplication.

7.6 The where statement and construct

It is often desired to perform an array operation only for certain elements, say those whose values are positive. The `where` statement provides this facility. A simple example is

```
where ( a > 1.0 ) a = 1.0/a    ! a is a real array
```

which reciprocates those elements of `a` that are greater than 1.0 and leaves the rest unaltered. The general form is

`where (`*logical-array-expr*`)` *array-variable* `=` *expr*

The logical array expression *logical-array-expr* is known as the **mask** and must have the same shape as *array-variable*. It is evaluated first and then just those elements of *expr* that correspond to elements of *logical-array-expr* that have the value true are evaluated and are assigned to the corresponding elements of *array-variable*. All other elements of *array-variable* are left unaltered. The assignment may be a defined assignment, provided that it is elemental (Section 7.9).

A single masking expression may be used for a sequence of array assignments all of the same shape. The simplest form of this construct is

```
where (logical-array-expr)
   array-assignments
end where
```

The masking expression *logical-array-expr* is first evaluated and then each array assignment is performed in turn, under the control of this mask. If any of these assignments affect entities in *logical-array-expr*, it is always the value obtained when the `where` statement is executed that is used as the mask.

The `where` construct may take the form

```
where (logical-array-expr)
   array-assignments
elsewhere
   array-assignments
end where
```

Here, the assignments in the first block of assignments are performed in turn under the control of *logical-array-expr* and then the assignments in the second block are performed in turn under the control of `.not.`*logical-array-expr*. Again, if any of these assignments affect entities in *logical-array-expr*, it is always the value obtained when the `where` statement is executed that is used as the mask.

A simple example of a `where` construct is

```
where (pressure <= 1.0)
   pressure = pressure + inc_pressure
   temp = temp + 5.0
elsewhere
   raining = .true.
end where
```

where `pressure`, `inc_pressure`, `temp`, and `raining` are arrays of the same shape.

If a `where` statement or construct masks an elemental function reference, the function is called only for the wanted elements. For example,

```
where ( a > 0 ) a = log(a)
```

(`log` is defined in Section 9.4) would not lead to erroneous calls of `log` for negative arguments.

This masking applies to all elemental function references except any that are within an argument of a non-elemental function reference. The masking does not extend to array arguments of such a function. In general, such arguments have a different shape so that masking would not be possible. For example, in the case

```
where (a > 0) a = a/sum(log(a))
```

(`sum` is defined in Section 9.13) the logarithms of each of the elements of `a` are summed and the statement will fail if they are not all positive.

If a non-elemental function reference or an array constructor is masked, it is fully evaluated before the masking is applied.

It is permitted to mask not only the `where` statement of the `where` construct, but also any `elsewhere` statement that it contains. All the masking expressions must be of the same

shape. A `where` construct may contain any number of masked `elsewhere` statements but at most one `elsewhere` statement without a mask, and if present this must be the final one. In addition, `where` constructs may be nested within one another; all the masking expressions of the nested construct must be of the same shape, and this must be the shape of all the array variables on the left-hand sides of the assignments within the construct.

A simple `where` statement such as that at the start of this section is permitted within a `where` construct and is interpreted as if it were the corresponding `where` construct containing one array assignment.

Finally, a `where` construct may be named in the same way as other constructs.

An example illustrating more complicated `where` constructs that are named is shown in Figure 7.6.

Figure 7.6 Nested `where` constructs, showing the masking.

```
assign_1: where (cond_1)
             ⋮                ! masked by cond_1
          elsewhere (cond_2)
             ⋮                ! masked by
             ⋮                ! cond_2.and..not.cond_1
assign_2:    where (cond_4)
                ⋮             ! masked by
                ⋮             ! cond_2.and..not.cond_1.and.cond_4
             elsewhere
                ⋮        ! masked by
                ⋮        ! cond_2.and..not.cond_1.and..not.cond_4
             end where assign_2
             ⋮
          elsewhere (cond_3) assign_1
             ⋮           ! masked by
             ⋮           ! cond_3.and..not.cond_1.and..not.cond_2
          elsewhere assign_1
             ⋮       ! masked by
             ⋮       ! not.cond_1.and..not.cond_2.and..not.cond_3
          end where assign_1
```

All the statements of a `where` construct are executed one by one in sequence, including the `where` and `elsewhere` statements. The masking expressions in the `where` and `elsewhere` statements are evaluated once and control of subsequent assignments is not affected by changes to the values of these expressions. Throughout a `where` construct there is a control

mask and a pending mask which change after the evaluation of each `where`, `elsewhere`, and `end where` statement, as illustrated in Figure 7.6.

7.7 Mask arrays

Logical arrays are needed for masking in `where` statements and constructs (Section 7.6), and they play a similar role in many of the array intrinsic functions (Chapter 9). Such arrays are often large, and there may be a worthwhile storage gain from using non-default logical types, if available. For example, some processors may use bytes to store elements of `logical(kind=1)` arrays, and bits to store elements of `logical(kind=0)` arrays. Unfortunately, there is no portable facility to specify such arrays, since there is no intrinsic function comparable to `selected_int_kind` and `selected_real_kind`.

Logical arrays are formed implicitly in certain expressions, usually as compiler-generated temporary variables. In

```
where (a > 0.0) a = 2.0 * a
```

or

```
if (any(a > 0.0)) then
```

(`any` is described in Section 9.13.1) the expression `a > 0.0` is a logical array. In such a case, an optimizing compiler can be expected to choose a suitable kind type parameter for the temporary array.

7.8 Pure procedures

In the description of functions in Section 5.10 we noted that, although it is permissible to write functions with side-effects, this is regarded as undesirable. It is possible for the programmer to assert that a procedure has no side-effects by adding the `pure` keyword to the `subroutine` or `function` statement. Declaring a procedure to be pure is an assertion that the procedure

i) if a function, does not alter any dummy argument;

ii) does not alter any part of a variable accessed by host or use association;

iii) contains no local variable with the `save` attribute (Section 8.10);

iv) performs no operation on an external file (Chapters 10 and 12);

v) contains no `stop` or `error stop` (Section 17.14) statement; and

vi) contains no image control statement (Section 17.13).

To ensure that these requirements are met and that a compiler can easily check that this is so, there are the following further rules:

i) Any dummy argument that is a procedure, and any procedure referenced, must be pure and have an explicit interface.

ii) The intent of a dummy argument must be declared unless it is a procedure or a pointer or has the `value` attribute, and this intent must be `in` in the case of a function.

iii) Any procedure internal to a pure procedure must be pure.

iv) A variable that is accessed by host or use association, is an intent `in` dummy argument, is a coindexed object (Section 17.4), or any part of such a variable must not be used in any way that could alter its value or pointer association status, or cause it to be the target of a pointer.

This last rule ensures that a local pointer cannot cause a side-effect.

The function in Figure 5.6 (Section 5.10) is pure, and this could be specified explicitly:

```
pure function distance(p, q)
```

An external or dummy procedure that is used as a pure procedure must have an interface block that specifies it as pure. However, the procedure may be used in other contexts without the use of an interface block or with an interface block that does not specify it as pure. This allows library procedures to be specified as pure without limiting them to being used as such.

Unlike pure functions, pure subroutines may have dummy arguments that have intent `out` or `inout` or the pointer attribute. The main reason now for allowing pure subroutines is to be able to use a defined assignment in a `do concurrect` statement (Section 7.17).[2] Their existence also gives the possibility of making subroutine calls from within pure functions.

All the intrinsic functions (Chapter 9) are pure, and can thus be referenced freely within pure procedures. Also, the elemental intrinsic subroutine `mvbits` (Section 9.10.5) is pure.

The `pure` attribute is given automatically to any procedure that has the `elemental` attribute (next section).

7.9 Elemental procedures

We have met already the notion of elemental intrinsic procedures (Section 7.4) – those with scalar dummy arguments that may be called with array actual arguments provided that the array arguments have the same shape (that is, provided all the arguments are conformable). For a function, the shape of the result is the shape of the array arguments. This feature also exists for non-intrinsic procedures. This requires the `elemental` prefix on the `function` or `subroutine` statement. For example, we could make the function `add_intervals` of Section 3.9 elemental, as shown in Figure 7.7. This is an aid to optimization on parallel processors.

Figure 7.7 An elemental function.

```
elemental function add_intervals(a,b)
   type(interval)             :: add_intervals
   type(interval), intent(in) :: a, b
   add_intervals%lower = a%lower + b%lower ! Production code
   add_intervals%upper = a%upper + b%upper ! would allow for
end function add_intervals                 ! roundoff.
```

Unless the `impure` prefix is used, see Section 7.10, an elemental procedure automatically has the `pure` attribute. In addition, any dummy argument must be a scalar variable that is not a coarray (Chapter 17), is not allocatable, and is not a pointer; and a function result must be

[2] Pure procedures were originally introduced to support the `forall` feature (Section B.3.1), now superseded by `do concurrent`.

a scalar variable that is not allocatable, is not a pointer, and does not have a type parameter defined by an expression other than a constant expression.

An interface block for an external procedure is required if the procedure itself is non-intrinsic and elemental. The interface must specify it as elemental. This is because the compiler may use a different calling mechanism in order to accommodate the array case efficiently. It contrasts with the case of pure procedures, where more freedom is permitted (see previous section).

For an elemental subroutine, if any actual argument is array valued, all actual arguments corresponding to dummy arguments with intent `inout` or `out` must be arrays. For example, we can make the subroutine `swap` of Figure 7.2 (Section 7.3) perform its task on arrays of any shape or size, as shown in Figure 7.8. Calling `swap` with an array and a scalar argument is obviously erroneous and is not permitted.

Figure 7.8 Elemental version of the subroutine of Figure 7.2.

```
elemental subroutine swap(a, b)
   real, intent(inout)  :: a, b
   real                 :: work
   work = a
   a = b
   b = work
end subroutine swap
```

If a generic procedure reference (Section 5.18) is consistent with both an elemental and a non-elemental procedure, the non-elemental procedure is invoked. For example, we might write versions of `add_intervals` (Figure 7.7) for arrays of rank one and rely on the elemental function for other ranks. In general, one must expect the elemental version to execute more slowly for a specific rank than the corresponding non-elemental version.

We note that a non-intrinsic elemental procedure may not be used as an actual argument. A procedure is not permitted to be both elemental and recursive.[3]

7.10 Impure elemental procedures

Elemental procedures that are pure are an aid to parallel evaluation. They also permit a single function to replace separate functions for each permutation of conformant ranks; for a procedure with two arguments, that is 46 separate procedures (16 cases where both arguments have the same rank, 15 where the first was scalar and the second is an array, and 15 where the first is an an array and the second was scalar).

The `impure` prefix on the procedure heading allows one to get this effect when the function is not pure. It processes array argument elements one by one in array element order. An example is shown in Figure 7.9. This example is impure in three ways: it counts the number of overflows in the global variable `overflow_count`, it logs each overflow on the external unit `error_unit`, and it terminates execution with `stop` when too many errors have been encountered.

[3]This restriction is removed in Fortran 2018, see Section 23.14.

Figure 7.9 An impure elemental function. The `write` statement is described in Chapter 11.

```
module safe_arithmetic
  integer :: max_overflows = 1000
  integer :: overflow_count = 0
contains
  impure elemental integer function square(n)
    use iso_fortran_env, only:error_unit
    integer, intent(in) :: n
    double precision, parameter :: sqrt_huge = &
                                   sqrt(real(huge(n), kind(0d0)))
    if (abs(n)>sqrt_huge) then
      write (error_unit,*) 'Overflow in square (', n, ')'
      overflow_count = overflow_count + 1
      if (overflow_count>max_overflows) stop '?Too many overflows'
      square = huge(n)
    else
      square = n**2
    end if
  end function
end module
```

Only the requirements relating to 'purity' (lack of side-effects) are lifted: the elemental requirements remain, that is:

- each dummy argument must be a scalar non-coarray dummy data object, must not have the `pointer` or `allocatable` attribute, and must either have specified intent or the `value` attribute;
- if the procedure is a function, its result must be scalar, must not have the `pointer` or `allocatable` attribute, and must not have a type parameter expression that depends on the value of a dummy argument, on the value of a deferred type parameter of a dummy argument, or on the bounds of a pointer or allocatable dummy array;
- in a reference to the procedure, all actual arguments must be conformable; and
- in a reference to the procedure, actual arguments corresponding to intent `out` and `inout` dummy arguments must either all be arrays or all be scalar.

7.11 Array elements

In Section 2.10 we restricted the description of array elements to simple cases. In general, an array element is a scalar of the form

part-ref [%part-ref] ...

where *part-ref* is

part-name [(*subscript-list*) *]*

and the last *part-ref* has a *subscript-list*. The number of subscripts in each list must be equal to the rank of the array or array component, and each subscript must be a scalar integer expression whose value is within the bounds of the dimension of the array or array component. To illustrate this, take the type

```
type triplet
   real                 :: u
   real, dimension(3)   :: du
   real, dimension(3,3) :: d2u
end type triplet
```

which was considered in Section 2.10. An array may be declared of this type:

```
type(triplet), dimension(10,20,30) :: tar
```

and

```
tar(n,2,n*n)         ! n of type integer
```

is an array element. It is a scalar of type `triplet` and

```
tar(n, 2, n*n)%du
```

is a real array with

```
tar(n, 2, n*n)%du(2)
```

as one of its elements.

If an array element is of type character, it may be followed by a substring reference:

(*substring-range*)

for example,

```
page (k*k) (i+1:j-5) ! i, j, k of type integer
```

By convention, such an object is called a substring rather than an array element.

Notice that it is the array *part-name* that the subscript list qualifies. It is not permitted to apply such a subscript list to an array designator unless the designator terminates with an array *part-name*. An array section, a function reference, or an array expression in parentheses must not be qualified by a subscript list.

7.12 Array subobjects

Array sections were introduced in Section 2.10 and provide a convenient way to access a regular subarray such as a row or a column of a rank-two array:

```
a(i, 1:n)   ! Elements 1 to n of row i
a(1:m, j)   ! Elements 1 to m of column j
```

For simplicity of description we did not explain that one or both bounds may be omitted when the corresponding bound of the array itself is wanted, and that a stride other than one may be specified:

```
a(i, :)       ! The whole of row i
a(i, 1:n:3)   ! Elements 1, 4, ... of row i
```

Another form of section subscript is a rank-one integer expression. All the elements of the expression must be defined with values that lie within the bounds of the parent array's subscript. For example, given an array `v` of extent 8,

```
v( [ 1, 7, 3, 2 ] )
```

is a section with elements `v(1)`, `v(7)`, `v(3)`, and `v(2)`, in this order. Such a subscript is called a **vector subscript**. If there are any repetitions in the values of the elements of a vector subscript, the section is called a **many–one section** because more than one element of the section is mapped onto a single array element. For example,

```
v( [ 1, 7, 3, 7 ] )
```

has elements 2 and 4 mapped onto `v(7)`. A many–one section must not appear on the left of an assignment statement because there would be several possible values for a single element. For instance, the statement

```
v( [ 1, 7, 3, 7 ] ) = [ 1, 2, 3, 4 ]     ! Invalid
```

is not allowed because the values 2 and 4 cannot both be stored in `v(7)`. The extent is zero if the vector subscript has zero size. Note that, with a sufficient repetition of subscript values, the section can actually be longer than the parent array.

When an array section with a vector subscript is an actual argument in a call of a non-elemental procedure, it is regarded as an expression and the corresponding dummy argument must not be defined or redefined and must not have intent `out` or `inout`. We expect compilers to make a copy as a temporary regular array on entry but to perform no copy back on return. Also, an array section with a vector subscript is not permitted to be a pointer target, since allowing them would seriously complicate the mechanism that compilers would otherwise have to establish for pointers. For similar reasons, such an array section is not permitted to be an internal file (Section 10.6).

In addition to the regular and irregular subscripting patterns just described, the intrinsic circular shift function `cshift` (Section 9.15.5) provides a mechanism that manipulates array sections in a 'wrap-round' fashion. This is useful in handling the boundaries of certain types of periodic grid problems, although it is subject to similar restrictions to those on vector subscripts. If an array `v(5)` has the value `[1,2,3,4,5]`, then `cshift(v, 2)` has the value `[3,4,5,1,2]`.

The general form of a subobject is

part-ref [%part-ref] ... [(*substring-range*) *]*

where *part-ref* now has the form

part-name [(*section-subscript-list*) *]*

where the number of section subscripts in each list must be equal to the rank of the array or array component. Each *section-subscript* is either a *subscript* (Section 7.11), a rank-one integer expression (vector subscript), or a *subscript-triplet* of the form

[lower] : *[upper] [* : *stride]*

where *lower*, *upper*, and *stride* are scalar integer expressions. If *lower* is omitted, the default value is the lower bound for this subscript of the array. If *upper* is omitted, the default value is the upper bound for this subscript of the array. If *stride* is omitted, the default value is one. The stride may be negative so that it is possible to take, for example, the elements of a row in reverse order by specifying a section such as

```
a(i, 10:1:-1)
```

The extent is zero if *stride* >0 and *lower* $>$ *upper*, or if *stride* <0 and *lower* $<$ *upper*. The value of *stride* must not be zero.

Normally, we expect the values of both *lower* and *upper* to be within the bounds of the corresponding array subscript. However, all that is required is that each value actually used to select an element is within the bounds. Thus,

```
a(1, 2:11:2)
```

is allowed even if the upper bound of the second dimension of `a` is only 10.

The *subscript-triplet* specifies a sequence of subscript values,

lower, *lower*+*stride*, *lower*+2×*stride*, ...

going as far as possible without going beyond *upper* (above it when *stride* >0 or below it when *stride* <0). The length of the sequence for the *i*th *subscript-triplet* determines the *i*th extent of the array that is formed.

The rank of a *part-ref* with a *section-subscript-list* is the number of vector subscripts and subscript triplets that it contains. So far in this section all the examples have been of rank one; by contrast, the ordinary array element

```
a(1,7)
```

is an example of a *part-ref* of rank zero, and the section

```
a(:,1:7)
```

is an example of a *part-ref* of rank two. The rank of a *part-ref* without a *section-subscript-list* is the rank of the object or component. A *part-ref* may be an array; for example,

```
tar%du(2)
```

for the array `tar` of Section 7.11 is an array section with elements `tar(1,1,1)%du(2)`, `tar(2,1,1)%du(2)`, `tar(3,1,1)%du(2)`, ... Being able to form sections in this way from arrays of derived type, as well as by selecting sets of elements, is a very useful feature of the language. A more prosaic example, given the specification

```
type(person), dimension(1:50) :: my_group
```

for the type `person` of Section 2.9, is the subobject `my_group%id` which is an integer array section of size 50.

Further, for the particular case of complex arrays, the `re` and `im` selectors can be applied to yield an array section comprising the real or imaginary part of each element of the array. For example,

```
complex :: x(n), y(n)
x%im = 2.0*y%im
```

Unfortunately, it is not permissible for more than one *part-ref* to be an array; for example, it is not permitted to write

```
tar%du   ! Invalid
```

for the array `tar` of Section 7.11. The reason for this is that if `tar%du` were considered to be an array, its element (1,2,3,4) would correspond to

```
tar(2,3,4)%du(1)
```

which would be too confusing a notation.

The *part-ref* with nonzero rank determines the rank and shape of the subobject. If any of its extents is zero, the subobject itself has size zero. It is called an array section if the final *part-ref* has a *section-subscript-list* or another *part-ref* has a nonzero rank.

A *substring-range* may be present only if the last *part-ref* is of type character and is either a scalar or has a *section-subscript-list*. By convention, the resulting object is called a section rather than a substring. It is formed from the unqualified section by taking the specified substring of each element. Note that, if `c` is a rank-one character array,

```
c(i:j)
```

is the section formed from elements `i` to `j`; if substrings of all the array elements are wanted, we may write the section

```
c(:)(k:l)
```

An array section that ends with a component name is also called a **structure component**. Note that if the component is scalar, the section cannot be qualified by a trailing subscript list or section subscript list. Thus, using the example of Section 7.11,

```
tar%u
```

is such an array section and

```
tar(1, 2, 3)%u
```

is a component of a valid element of `tar`. The form

```
tar%u(1, 2, 3)  ! not permitted
```

is not allowed.

Additionally, a *part-name* to the right of a *part-ref* with nonzero rank must not have the `allocatable` or `pointer` attribute. This is because such an object would represent an array whose elements were independently allocated and would require a very different implementation mechanism from that needed for an ordinary array. For example, consider the array

```
type(entry), dimension(n) :: rows  ! n of type integer
```

for the type `entry` defined in Figure 2.3. If we were allowed to write the object `rows%next`, it would be interpreted as another array of size `n` and type `entry`, but its elements are likely to be stored without any regular pattern (each having been separately given storage by an `allocate` statement) and indeed some will be null if any of the pointers are disassociated. Note that there is no problem over accessing individual pointers such as `rows(i)%next`.

7.13 Arrays of pointers

Although arrays of pointers as such are not allowed in Fortran, the equivalent effect can be achieved by creating a type containing a pointer component. This is useful when constructing a linked list that is more complicated than the chain described in Section 2.12. For instance, if a variable number of links are needed at each entry, the recursive type `entry` of Figure 2.3 might be expanded to the pair of types:

```
type ptr
   type(entry), pointer :: point
end type ptr
type entry
   real                 :: value
   integer              :: index
   type(ptr), pointer   :: children(:)
end type entry
```

After appropriate allocations and pointer associations, it is then possible to refer to the index of child `j` of `node` as

```
node%children(j)%point%index
```

This extra level of indirection is necessary because the individual elements of `children` do not, themselves, have the `pointer` attribute – this is a property only of the whole array. For example, we can take two existing nodes, say `a` and `b`, each of which is a tree root, and make a big tree thus

```
tree%children(1)%point => a
tree%children(2)%point => b
```

which would not be possible with the original type `entry`.

7.14 Pointers as aliases

If an array section without vector subscripts, such as

```
table(m:n, p:q)
```

is wanted frequently while the integer variables `m`, `n`, `p`, and `q` do not change their values, it is convenient to be able to refer to the section as a named array such as

```
window
```

Such a facility is provided in Fortran by pointers and the pointer assignment statement. Here, `window` would be declared thus

```
real, dimension(:, :), pointer :: window
```

and associated with `table`, which must of course have the `target` or `pointer` attribute,[4] by the execution of the statement

```
window  => table(m:n, p:q)
```

If, later on, the size of `window` needs to be changed, all that is needed is another pointer assignment statement. Note, however, that the subscript bounds for `window` in this example are `(1:n-m+1, 1:q-p+1)` since they are as provided by the functions `lbound` and `ubound` (Section 9.14.2).

The facility provides a mechanism for subscripting or sectioning arrays such as

[4]The `associate` construct (Section 15.6) provides a means of achieving this in a limited scope without the need for the `target` or `pointer` attribute.

```
tar%u
```

where `tar` is an array and `u` is a scalar component, discussed in Section 7.11. Here we may perform the pointer association

```
taru => tar%u
```

if `taru` is a rank-three pointer of the appropriate type. Subscripting as in

```
taru(1, 2, 3)
```

is then permissible. Here the subscript bounds for `taru` will be those of `tar`.

7.15 Pointer assignment

There are two further facilities concerning the array pointer assignment statement. The first is that it is possible to set the desired lower bounds to any value. This can be desirable in situations like the following. Consider

```
real, target  :: annual_rainfall(1700:2003)
real, pointer :: rp1(:), rp2(:)
⋮
rp1 => annual_rainfall
rp2 => annual_rainfall(1800:1856)
```

The bounds of `rp1` will be `(1700:2003)`; however, those of `rp2` will be `(1:57)`. To be able to have a pointer to a subsection of an array have the appropriate bounds, they may be set on the pointer assignment as follows:

```
rp2(1800:) => annual_rainfall(1800:1856)
```

This statement will set the bounds of `rp2` to `(1800:1856)`.

The second facility for array pointer assignment is that the target of a multi-dimensional array pointer may be one-dimensional or simply contiguous (Section 7.18.2). The syntax is similar to that of the lower-bounds specification above, except that in this case one specifies each upper bound as well as each lower bound. The elements of the array pointer, in array element order, are associated with the leading elements of the target array. This can be used, for example, to provide a pointer to the diagonal of an array:

```
real, pointer :: base_array(:), matrix(:,:), diagonal(:)
allocate (base_array(n*n))
matrix(1:n, 1:n) => base_array
diagonal => base_array(::n+1)
```

After execution of the pointer assignments, `diagonal` is now a pointer to the diagonal elements of `matrix`.

7.16 Array constructors

The syntax that we introduced in Section 2.10 for array constants may be used to construct more general rank-one arrays. The general form of an *array-constructor* is

(/ *[type-spec* :: *] array-constructor-value-list* /)

or

[*[type-spec* :: *] array-constructor-value-list*]

where each *array-constructor-value* is one of *expr* or *constructor-implied-do*, and the *type-spec* (if it appears) is the type name followed by the type parameter values in parentheses, if any, for both intrinsic and derived types. The form with square brackets is to make it easier to match parentheses; note that you cannot mix them, so (/...] and [.../) are not allowed.

The array thus constructed is of rank one with its sequence of elements formed from the sequence of scalar expressions and elements of the array expressions in array element order. A *constructor-implied-do* has the form

(*array-constructor-value-list*, *variable* = $expr_1$, $expr_2$ [, $expr_3$])

where *variable* is a named integer scalar variable, and $expr_1$, $expr_2$, and $expr_3$ are scalar integer expressions. Its interpretation is as if the *array-constructor-value-list* had been written

$$\max(\ (expr_2 - expr_1 + expr_3) \div expr_3,\ 0\ )$$

times, with *variable* replaced by $expr_1$, $expr_1 + expr_3, \ldots$, as for the `do` construct (Section 4.4). A simple example is

```
(/ (i,i=1,10) /)
```

which is equal to

```
(/ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 /)
```

Note that the syntax permits nesting of one *constructor-implied-do* inside another, as in the example

```
(/ ((i,i=1,3), j=1,3) /)
```

which is equal to

```
(/ 1, 2, 3, 1, 2, 3, 1, 2, 3 /)
```

and the nesting of structure constructors within array constructors (and vice versa); for instance, for the type in Section 7.5,

```
(/ (matrix(0.0), i = 1, limit) /)
```

The sequence may be empty, in which case a zero-sized array is constructed. The scope of the *variable* is the *constructor-implied-do*. Other statements, or even other parts of the array constructor, may refer to another variable having the same name. The value of the other variable is unaffected by execution of the array constructor and is available except within the *constructor-implied-do*.

An array of rank greater than one may be constructed from an array constructor by using the intrinsic function `reshape` (Section 9.15.3). For example,

```
reshape( source = [ 1,2,3,4,5,6 ], shape = [ 2,3 ] )
```

has the value

1 3 5
2 4 6

If the array constructor does not begin with *type-spec* `::`, its type and type parameters are those of the first *expr*, and each *expr* must have the same type and type parameters. If every *expr*, $expr_1$, $expr_2$, and $expr_3$ is a constant expression (Section 8.4), the array constructor is a constant expression.

If an array constructor begins with *type-spec* `::`, its type and type parameters are those specified by the *type-spec*. In this case, the array constructor values may have any type and type parameters (including character length) that are assignment-compatible with the specified type and type parameters, and the values are converted to that type by the usual assignment conversions.

Here are some examples:

```
[ character(len=33) :: 'the good', 'the bad', 'and', &
                       'the appearance-challenged' ]
[ complex(kind(0d0)) :: 1, (0,1), 3.14159265358979323846264338327d0 ]

[ matrix(kind=kind(0.0), n=10, m=20) :: ] ! zero-sized array
```

7.17 The do concurrent construct

A further form of the `do` construct, the `do concurrent` construct, is provided to help improve performance by enabling parallel execution of the loop iterations. The basic idea is that by using this construct the programmer asserts that there are no interdependencies between loop iterations. The effect is similar to that of various compiler-specific directives such as '`!dec$ivdep`'; such directives have been available for a long time, but often have slightly different meanings on different compilers.

Use of `do concurrent` has a long list of requirements which can be grouped into 'limitations' on what may appear within the construct and 'guarantees' by the programmer that the computation has certain properties (essentially, no dependencies) that enable parallelization. Note that in this context parallelization does not necessarily require multiple processors, or even if multiple processors are available, that they will be used: other optimizations that improve single-threaded performance are also enabled by these properties, including vectorization, pipelining, and other possibilities for overlapping the execution of instructions from more than one iteration on a single processor.

The form of the `do concurrent` statement is

`do` *[,]* `concurrent` `(` *[type-spec* `::` *]* *index-spec-list [, scalar-mask-expr]* `)`

where *type-spec* (if present) specifies the type and kind of the index variables, and *index-spec-list* is a list of index specifications of the form

index-variable-name `=` *initial-value* `:` *final-value [* `:` *step-value]*

as in

```
do concurrent (i=1:n, j=1:m)
```

Each *index-variable-name* is local to the loop, so has no effect on any variable with the same name that might exist outside the loop; however, if *type-spec* is omitted, it has the type

and kind it would have if it were such a variable. In either case, it must be scalar and have `integer` type. Each *initial-value*, *final-value*, and *step-value* is a scalar integer expression.

The optional *scalar-mask-expr* is of type `logical`; if it appears, only those iterations that satisfy the condition are executed. That is,

```
do concurrent (i=1:n, j=1:m, i/=j)
  ⋮
end do
```

has exactly the same meaning as

```
do concurrent (i=1:n, j=1:m)
  if (i/=j) then
    ⋮
  end if
end do
```

A simple example of a `do concurrent` construct is

```
do concurrent (i=1:n)
  a(i, j) = a(i, j) + alpha*b(i, j)
end do
```

The following items are all prohibited within a `do concurrent` construct (and the compiler is required to detect these except for `advance=`):

- a statement that would terminate the `do concurrent` construct: that is, a `return` statement (Section 5.8), a branch (for example, `go to` or `err=`) with a label that is outside the construct, an `exit` statement that would exit from the `do concurrent` construct, or a `cycle` statement that names an outer `do` construct;
- an image control statement (Chapter 17);
- a reference to a procedure that is not pure (Section 7.8);
- a reference to one of the procedures `ieee_get_flag`, `ieee_set_halting_mode`, or `ieee_get_halting_mode` from the intrinsic module `ieee_exceptions` (Section 18.8); and
- an input/output statement with an `advance=` specifier (Section 10.11).

The above are all prohibited because they are either impossible or extremely difficult to use without breaking the 'no interdependencies between iterations' rule.

By using `do concurrent` the programmer guarantees that executing the loop iterations in a different order would not change the results (except possibly for the order of records output to a sequential file); in particular:

- any variable referenced is either previously defined in the same iteration, or its value is not affected by any other iteration;
- any pointer that is referenced is either previously pointer associated in the same iteration, or does not have its pointer association changed by any other iteration;

- any allocatable object that is allocated or deallocated by an iteration shall not be used[5] by any other iteration, unless every iteration that uses the object first allocates it and finally deallocates it;
- records (or positions) in a file are not both written by one iteration and read back by another iteration (Chapter 10).

If records are written to a sequential file (Chapter 10) by more than one iteration of the loop, the ordering between the records written by different iterations is indeterminate. That is, the records written by one iteration might appear before the records written by the other, after the records written by the other, or be interspersed.

Furthermore, when execution of the construct has completed,

- any variable whose value is affected by more than one iteration becomes undefined on termination of the loop; and
- any pointer whose association status is changed by more than one iteration has an association status of undefined.

Note that any ordinary `do` loop that satisfies the limitations and which obviously has the required properties can be parallelized, so use of `do concurrent` is not necessary for parallel execution. In fact, a compiler that parallelizes `do concurrent` is likely to treat it as a request that it should parallelize that loop; if the loop iteration count is very small, this could result in worse performance than an ordinary `do` loop due to the overhead of initiating parallel threads of execution. Thus, even when the programmer-provided guarantees are trivially derived from the loop body itself, `do concurrent` is still useful for

- indicating to the compiler that this is likely to have a high enough iteration count to make parallelization worthwhile;
- using the compiler to enforce the prohibitions (e.g. no calls to impure procedures);
- documenting the parallelizability for code reading and maintenance; and
- as a crutch to compilers whose analysis capabilities are limited.

7.18 Performance-oriented features

7.18.1 The contiguous attribute

It is expected that most arrays will be held in contiguous memory, but there are exceptions such as these array sections:

```
vector(::2)     ! all the odd-numbered elements
vector(10,1,-1) ! reverse order
cxarray%re      ! the real parts of a complex array
```

(where the third line uses the syntax of Section 2.9). A pointer or assumed-shape array that is not contiguous can come about by association with an array section that is not contiguous, and the compiler has to allow for this. The `contiguous` attribute is an attribute for pointer and assumed-shape dummy arrays. For an array pointer, it restricts its target to being contiguous.

[5] That is, referenced, defined, allocated, deallocated, or have any dynamic property enquired about.

For an assumed-shape array, it specifies that if the corresponding actual argument is not contiguous, copy-in copy-out is used make the dummy argument contiguous.

Knowing that an array is contiguous in this sense simplifies array traversal and array element address calculations, potentially improving performance. Whether this improvement is significant depends on the fraction of time spent performing traversal and address calculation operations; in some programs this time is substantial, but in many cases it is insignificant in the first place.

Traditionally, the Fortran standard has shied away from specifying whether arrays are contiguous in the sense of occupying sequential memory locations with no intervening unoccupied spaces. In the past this tradition has enabled high-performance multi-processor implementations of the language, but the `contiguous` attribute is a move towards more specific hardware limitations. Although contiguous arrays are described only in terms of language restrictions and not in terms of the memory hardware, the interaction between these and interoperability with the C language (Chapter 19) means that these arrays will almost certainly be stored in contiguous memory locations.

Any of the following arrays are considered to be contiguous by the standard:

- an array with the `contiguous` attribute;
- a whole array (named array or array component without further qualification) that is not a pointer or assumed-shape;
- an assumed-shape array that is argument associated with an array that is contiguous;
- an array allocated by an `allocate` statement;
- a pointer associated with a contiguous target; or
- a nonzero-sized array section provided that
  - its base object is contiguous;
  - it does not have a vector subscript;
  - the elements of the section, in array element order, are elements of the base object that are consecutive in array element order;
  - if the array is of type character and a substring selector appears, the selector specifies all of the characters of the string;
  - it is not a component of an array; and
  - it is not the real or imaginary part of an array of type complex.

A subobject (of an array) is definitely not contiguous if all of these conditions apply:

- it (the subobject) has two or more elements;
- its elements in array element order are not consecutive in the elements of the original array;
- it is not a zero-length character array; and
- it is not of a derived type with no ultimate components other than zero-sized arrays and zero-length character strings.

Whether an array that is in neither list is contiguous or not is compiler-specific.

The `contiguous` attribute can be specified with the `contiguous` keyword on a type declaration statement, for example

```
subroutine s(x)
  real, contiguous :: x(:,:)
  real, pointer, contiguous :: column(:)
```

It can also be specified by the `contiguous` statement, which has the form

`contiguous` *[::] object-name-list*

Contiguity can be tested with the inquiry function `is_contiguous` (Section 9.14.1).

Figure 7.10 Using `is_contiguous` before using `c_loc`.

```
subroutine process(x)
  real(c_float), target :: x(:)
  interface
    subroutine c_routine(a, nbytes)
      use iso_c_binding
      type(c_ptr), value       :: a
      integer(c_size_t), value :: nbytes
    end subroutine
  end interface
  ⋮
  if (is_contiguous(x)) then
    call c_routine(c_loc(x), c_sizeof(x))
  else
    stop 'x needs to be contiguous'
  end if
end subroutine
```

Arrays in C are always contiguous, so referencing the intrinsic function `c_loc` (Section 19.3) is permitted for any target that is contiguous (at execution time). The example in Figure 7.10 uses `is_contiguous` to check that it is being asked to process a contiguous object, and produces an error message if it is not. It also makes use of the `c_sizeof` function to calculate the size of `x` in bytes (see Section 19.7).

There is also the concept of **simply contiguous**; that is, not only is the object contiguous, but it can be seen to be obviously so at compilation time. Unlike 'being contiguous', this is completely standardized. This is further discussed in the next section.

When dealing with `contiguous` assumed-shape arrays and array pointers, it is important to keep in mind the various run-time requirements and restrictions. For assumed-shape arrays, the `contiguous` attribute makes no further requirements on the program: if the actual argument is not contiguous, a local copy is made on entry to the procedure, and any changes to its value are copied back to the actual argument on exit. Depending on the number and manner of the references to the array in the procedure, the cost of copying can be higher than any putative performance savings given by the `contiguous` attribute. For example, in

```
complex function f(v1, v2, v3)
  real, contiguous, intent(in) :: v1(:), v2(:), v3(:)
  f = cmplx(sum(v1*v2*v3))**(-size(v1))
end function
```

since the arrays are only accessed once, if any actual argument is discontiguous this will almost certainly perform much worse than if the `contiguous` attribute were not present.

For array pointers, the `contiguous` attribute has a run-time requirement that it be associated only with a contiguous target (via pointer assignment). However, it is the programmer's responsibility to check this, or to 'know' that the pointer will never become associated with a discontiguous section. (Such knowledge is prone to becoming false in the course of program maintenance, so checking on each pointer assignment is recommended.) Similar comments apply to the use of the `c_loc` function on an array that might not be contiguous. If these requirements are violated, the program will almost certainly produce incorrect answers with no indication of the failure.

7.18.2 Simply contiguous array designators

A **simply contiguous** array designator is, in principle, a designator that not only describes an array (or array section) that is contiguous, but one that can easily be seen at compilation time to be contiguous. Whether a designator is simply contiguous does not depend on the value of any variable.

A simply contiguous array can be used in the following ways:

- as the target of a rank-remapping pointer assignment (that is, associating a pointer with a target of a different rank, see Section 7.15);
- as an actual argument corresponding to a dummy argument that is not an assumed-shape array or which is an assumed-shape array with the `contiguous` attribute, when both have either the `asynchronous` or `volatile` attribute;
- as an actual argument corresponding to a dummy pointer with the `contiguous` attribute (this also requires that the actual argument have the `pointer` or `target` attribute).

The example in Figure 7.11 'flattens' the matrix `a` into a simple vector, and then uses that to associate another pointer with the diagonal of the matrix.

Figure 7.11 Diagonal of contiguous matrix.

```
real, target  :: a(n, m)
real, pointer :: a_flattened(:), a_diagonal(:)
a_flattened(1:n*m) => a
a_diagonal         => a_flattened(::n+1)
```

In Figure 7.12, copy-in copy-out must be avoided in the call of `start_bufferin` because it will start an asynchronous input operation to read values into the array and reading will continue after return. Because both `x` and `y` are simply contiguous, copy-in copy-out is avoided.

Another example of the use of simply contiguous to enforce contiguity of an actual argument is explained in Section 7.18.3.

Also, for argument association with a dummy coarray (see Section 17.9) that is an array with the `contiguous` attribute or an array that is not assumed-shape, the actual argument is required to be simply contiguous in order to avoid any possibility of copy-in copy-out

Figure 7.12 Contiguous buffer for asynchronous input/output. Asynchronous input/output is described in Section 10.15.

```
interface
  subroutine start_bufferin(a, n)
    integer, intent(in)             :: n
    real, intent(out), asynchronous :: a(n)
  end subroutine
end interface
real, asynchronous :: x(n), y(n)
⋮
call start_bufferin(x)
⋮
call start_bufferin(y)
```

occurring. Unfortunately, the Fortran standard does not require detection of the violation of this rule, which means that a program that breaks it might crash or produce wrong answers without any warning.

Also, when a simply contiguous array with the `target` attribute and not the `value` attribute (Section 19.8) is used as the actual argument corresponding to a dummy argument that has the `target` attribute and is an assumed-shape array with the `contiguous` attribute or is an explicit-shape array,

- a pointer associated with the actual argument becomes associated with the dummy argument on invocation of the procedure; and
- when execution of the procedure completes, pointers in other scopes that were associated with the dummy argument are associated with the actual argument.

However, we do not recommend using this complicated fact, as it is difficult to understand and program maintenance is quite likely to break one of the essential conditions for its applicability.

An array designator is simply contiguous if and only if it is

- a whole array that has the `contiguous` attribute;
- a whole array that is not an assumed-shape array or array pointer; or
- a section of a simply contiguous array that
  - is not the real or imaginary part of a complex array (see Section 2.9);
  - does not have a substring selector;
  - is not a component of an array; and
  - either does not have a *section-subscript-list*, or has a *section-subscript-list* which specifies a simply contiguous section.

A *section-subscript-list* specifies a simply contiguous section if and only if

- it does not have a vector subscript;

- all but the last *subscript-triplet* is a colon;
- the last *subscript-triplet* does not have a stride; and
- no *subscript-triplet* is preceded by a *section-subscript* that is a subscript.

An array variable is simply contiguous if and only if it is a simply contiguous array designator or a reference to a function that returns a pointer with the `contiguous` attribute.

7.18.3 Automatic pointer targetting

An actual argument with the `target` attribute is permitted to correspond to a dummy pointer with the intent `in` attribute. This is illustrated by Figure 7.13; in this case, the automatic

Figure 7.13 Convenient automatic targetting. The `protected` attribute is described in Section 8.6.2.

```
module m
  real, pointer, protected :: parameter_list(:)
contains
  subroutine set_params(list)
    real, pointer, intent(in) :: list(:)
    parameter_list => list
  end subroutine
  ⋮
end module
⋮
subroutine solve(problem, args)
  use m
  real, target :: args(:)
  ⋮
  call set_params(args)
  ⋮
end subroutine
```

targetting is only being used for convenience, merely saving the hassle of creating a local pointer, pointing it at `args`, and passing the local pointer to `set_params`.

However, automatic targetting can also be used to enforce contiguity requirements; if a dummy pointer has the `contiguous` attribute, the actual argument must be simply contiguous (see Section 7.18.2). This means that the user can be sure that no unintended copying, by a copy-in copy-out argument-passing mechanism, is taking place. This is illustrated by Figure 7.14, which requires a contiguous array to be used for buffering operations. A call to `set_buffer` with an argument that is not simply contiguous would produce an error at compile time.

Figure 7.14 Automatic targetting of contiguous array.

```
module buffer_control
  character(:), contiguous, pointer, protected :: buffer(:)
contains
  subroutine set_buffer(charbuf)
    character(*), pointer, intent(in), contiguous :: charbuf(:)
    buffer => charbuf
  end subroutine
end module
⋮
character, allocatable, target :: mybuf(:)
⋮
allocate (mybuf(n))
call set_buffer(mybuf)
```

7.19 Summary

We have explained that arrays may have zero size and that no special rules are needed for them. Storage for an array may be allocated automatically on entry to a procedure and automatically deallocated on return. Functions may be array valued either through the mechanism of an elemental reference that performs the same calculation for each array element, or through the truly array-valued function. Elemental procedures may be pure or impure. Array assignments may be masked through the use of the `where` statement and construct. Structure components may be arrays if the parent is an array or the component is an array, but not both. A subarray may either be formulated directly as an array section, or indirectly by using pointer assignment to associate it with a pointer. An array may be constructed from a sequence of expressions. A logical array may be used as a mask.

The intrinsic functions are an important part of the array features and will be described in Chapter 9.

We conclude this chapter with a complete program, see Figures 7.15 and 7.16, that illustrates the use of array expressions, array assignments, allocatable arrays, automatic arrays, and array sections. The module `linear` contains a subroutine for solving a set of linear equations, and this is called from a main program that prompts the user for the problem and then solves it.

Figure 7.15 First part of a module for solving a set of linear equations; `size` is described in Section 9.14.2 and `maxloc` is described in Section 9.16.

```
module linear
   integer, parameter, public :: kind=selected_real_kind(10)
   public :: solve

contains
   subroutine solve(a, piv_tol, b, ok)
   ! arguments
      real(kind), intent(inout), dimension(:,:) :: a
                        ! The matrix a.
      real(kind), intent(in) :: piv_tol
                        ! Smallest acceptable pivot.
      real(kind), intent(inout), dimension(:) :: b
                        ! The right-hand side vector on
                        ! entry. Overwritten by the solution.
      logical, intent(out)      :: ok
                        ! True after a successful entry
                        ! and false otherwise.

   ! Local variables
      integer :: i     ! Row index.
      integer :: j     ! Column index.
      integer :: n     ! Matrix order.
      real(kind), dimension(size(b)) :: row
                        ! Automatic array needed for workspace;
      real(kind) :: element ! Workspace variable.

      n = size(b)
      ok = size(a, 1) == n .and. size(a, 2) == n
      if (.not.ok) then
         return
      end if

      do j = 1, n

!     Update elements in column j.
         do i = 1, j - 1
             a(i+1:n, j) = a(i+1:n, j) - a(i,j) * a(i+1:n, i)
         end do

!     Find pivot and check its size
         i = maxloc(abs(a(j:n, j)), dim=1) + j - 1
         if (abs(a(i, j)) < piv_tol) then
            ok = .false.
            return
         end if
```

Figure 7.16 Second part of Figure 7.15 module and a program that uses it. The edit descriptors used in the `write` statements are described in Chapter 11.

```
!     If necessary, apply row interchange
         if (i/=j) then
            row = a(j, :); a(j, :) = a(i, :); a(i, :) = row
            element = b(j); b(j) = b(i); b(i) = element
         end if

!     Compute elements j+1 : n of j-th column.
         a(j+1:n, j) = a(j+1:n, j)/a(j, j)
      end do

!  Forward substitution
      do i = 1, n-1
          b(i+1:n) = b(i+1:n) - b(i)*a(i+1:n, i)
      end do

!   Back-substitution
      do j = n, 1, -1
         b(j) = b(j)/a(j, j)
         b(1:j-1) =  b(1:j-1) - b(j)*a(1:j-1, j)
      end do
   end subroutine solve
end module linear

program main
   use linear
   integer :: i, n
   real(kind), allocatable :: a(:, :), b(:)
   logical :: ok

   print *, ' Matrix order?'
   read *,  n
   allocate ( a(n, n), b(n) )
   do i = 1, n
      write (*, '(a, i2, a)') ' Elements of row ', i, ' of a?'
      read *, a(i,:)
      write (*, '(a, i2, a)') ' Component ', i, ' of b?'
      read *, b(i)
   end do

   call solve(a, maxval(abs(a))*1.0e-10, b, ok)
   if (ok) then
      write (*, '(/,a,/,(5f12.4))') ' Solution is', b
   else
      print *, ' The matrix is singular'
   end if
end program main
```

Exercises

1. Given the array declaration

```
real, dimension(50,20) :: a
```

write array sections representing

i) the first row of `a`;

ii) the last column of `a`;

iii) every second element in each row and column;

iv) as for (iii) in reverse order in both dimensions;

v) a zero-sized array.

2. Write a `where` statement to double the value of all the positive elements of an array `z`.

3. Write an array declaration for an array `j` which is to be completely defined by the statement

```
j = (/ (3, 5, i=1,5), 5,5,5, (i, i =  5,3,-1 ) /)
```

4. Classify the following arrays:

```
subroutine example(n, a, b)
   real, dimension(n, 10) :: w
   real                   :: a(:), b(0:)
   real, pointer          :: d(:, :)
```

5. Write a declaration and a pointer assignment statement suitable to reference as an array all the third elements of component `du` in the elements of the array `tar` having all three subscript values even (Section 7.11).

6. Given the array declarations

```
integer, dimension(100, 100), target :: l, m, n
integer, dimension(:, :), pointer    :: ll, mm, nn
```

rewrite the statements

```
l(j:k+1, j-1:k) = l(j:k+1, j-1:k) + l(j:k+1, j-1:k)
l(j:k+1, j-1:k) = m(j:k+1, j-1:k) + n(j:k+1, j-1:k) + n(j:k+1, j:k+1)
```

as they could appear following execution of the statements

```
ll => l(j:k+1, j-1:k)
mm => m(j:k+1, j-1:k)
nn => n(j:k+1, j-1:k)
```

7. Complete Exercise 1 of Chapter 4 using array syntax instead of `do` constructs.

8. Write a module to maintain a data structure consisting of a linked list of integers, with the ability to add and delete members of the list, efficiently.

9. Write a module that contains the example in Figure 7.5 (Section 7.5) as a module procedure and supports the defined operations and assignments that it contains.

8. Specification statements

8.1 Introduction

In the preceding chapters we have learnt the elements of the Fortran language, how they may be combined into expressions and assignments, how we may control the logic flow of a program, how to divide a program into manageable parts, and have considered how arrays may be processed. We have seen that this knowledge is sufficient to write programs, when combined with rudimentary `read` and `print` statements and with the `end` statement.

In Chapters 2 to 7 we met some specification statements when declaring the type and other properties of data objects, but to ease the reader's task we did not always explain all the available options. In this chapter we fill this gap. To begin with, however, it is necessary to recall the place of specification statements in a programming language. A program is processed by a computer in stages. In the first stage, compilation, the source code (text) of the program is read by a program known as a **compiler** which analyses it, and generates files containing **object code**. Each program unit of the complete program is usually processed separately. The object code is a translation of the source code into a form which can be understood by the computer hardware, and contains the precise instructions as to what operations the computer is to perform. Using these files, an executable program is constructed. The final stage consists of the execution, whereby the coded instructions are performed and the results of the computations made available.

During the first stage, the compiler requires information about the entities involved. This information is provided at the beginning of each program unit or subprogram by specification statements. The description of most of these is the subject of this chapter. The specification statements associated with procedure interfaces, including interface blocks and the `interface` statement and also the `external` statement, were explained in Chapter 5. The `intrinsic` statement is explained in Section 9.1.3.

8.2 Implicit typing

Many programming languages require that all typed entities have their types specified explicitly. Any data entity that is encountered in an executable statement without its type having been declared will cause the compiler to indicate an error. Fortran also has implicit typing: an entity is assigned a type according to the initial letter of its name unless it is explicitly typed by appearing in a type declaration statement or being accessed by use or host association. The default is that an entity whose name begins with one of the letters `i`, `j`, ...,

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

`n` is of type default integer, and an entity whose name begins with one of the letters `a`, `b`, ..., `h`, or `o`, `p`, ..., `z` is of type default real.[1]

Implicit typing can lead to program errors; for instance, if a variable name is misspelt, the misspelt name will give rise to a separate variable which, if used, can lead to unforeseen consequences. For this reason, we recommend that implicit typing be avoided. The statement

```
implicit none
```

avoids implicit typing in a scoping unit.[2] It may be preceded within the scoping unit only by `use` (and `format`) statements. A scoping unit without an `implicit` statement has the same implicit typing (or lack of it) as its host. For example, a single `implicit none` statement in a module avoids implicit typing in the module and all the scoping units it contains.

8.3 Named constants

Inside a program, we often need to define a constant or set of constants. For instance, in a program requiring repeated use of the speed of light, we might use a real variable `c` that is given its value by the statement

```
c = 2.99792458
```

A danger in this practice is that the value of `c` may be overwritten inadvertently, for instance because another programmer reuses `c` as a variable to contain a different quantity, failing to notice that the name is already in use.

It might also be that the program contains specifications such as

```
real    :: x(10), y(10), z(10)
integer :: mesh(10, 10), ipoint(100)
```

where all the dimensions are 10 or 10^2. Such specifications may be used extensively, and 10 may even appear as an explicit constant, say as a parameter in a `do`-construct which processes these arrays:

```
do i = 1, 10
```

Later, it may be realized that the value 20 rather than 10 is required, and the new value must be substituted everywhere the old one occurs, an error-prone undertaking.

Yet another case was met in Section 2.6, where named constants were needed for kind type parameter values.

In order to deal with all of these situations, Fortran contains what are known as named constants. These may never appear on the left-hand side of an assignment statement, but may be used in expressions in any way in which a literal constant may be used. A type declaration statement may be used to specify such a constant:

```
real, parameter :: c = 2.99792458
```

The value is protected, as `c` is now the name of a constant and may not be used as a variable name in the same scoping unit. Similarly, we may write

[1] See Section A.8 for means of specifying other mappings between the letters and types.

[2] This statement is enhanced in Fortran 2018, see Section 23.2.

```
integer, parameter :: length = 10
real               :: x(length), y(length), z(length)
integer            :: mesh(length, length), ipoint(length**2)
⋮
do i = 1, length
```

which has the clear advantage that in order to change the value of 10 to 20 only a single line need be modified, and the new value is then correctly propagated.

A named constant may be an array, as in the case

```
real, dimension(3), parameter :: field = [ 0.0, 10.0, 20.0 ]
```

However, it is not necessary to declare the shape in advance: the shape may be taken from the value. This is called an **implied-shape array**, and is specified by an asterisk as upper bound, for example,

```
character, parameter :: vowels(*) = [ 'a', 'e', 'i', 'o', 'u' ]
```

For an array of rank greater than one, the `reshape` function described in Section 9.15.3 must be applied. If the array has implied shape, an asterisk must be specified for each upper bound. For example

```
integer, parameter :: powers(0:*,*) = &
reshape( [ 0, 1, 2, 3, 0, 1, 4, 9, 0, 1, 8, 27 ], [ 4, 3 ] )
```

declares `powers` to have the bounds `(0:3, 1:3)`.

Similarly, the length of a scalar named constant of type `character` may be specified as an asterisk and taken directly from its value, which obviates the need to count the length of a character string, making modifications to its definition much easier. An example of this is

```
character(len=*), parameter :: string = 'No need to count'
```

Unfortunately, counting is needed when a character array is defined using an array constructor without a type specifier, since all the elements must have the same length:

```
character(len=7), parameter, dimension(3) ::          &
                  c=['Cohen  ', 'Metcalf', 'Reid   ']
```

would not be correct without the two blanks in `'Cohen  '` and the three in `'Reid   '`. This need can be circumvented by using a *type-spec*, as shown in Section 7.16; but that has the possibility of truncation if the wrong length is specified in the *type-spec*.

A named constant may be of derived type, as in the case

```
type(posn), parameter :: a = posn(1.0,2.0,0)
```

for the type

```
type posn
   real    :: x, y
   integer :: z
end type posn
```

Note that a subobject of a constant need not necessarily have a constant value. For example, if `i` is an integer variable, `field(i)` may have the value 0.0, 10.0, or 20.0. Note also that a constant may not be a pointer, allocatable object, dummy argument, or function result, since these are always variables. However, it may be of a derived type with an allocatable component that is unallocated or a pointer component that is disassociated. Clearly, because such a component is part of a constant, it is not permitted to be allocated or pointer assigned.

The `parameter` attribute is an important means whereby constants may be protected from overwriting, and programs modified in a safe way. It should be used for these purposes on every possible occasion.

8.4 Constant expressions

In an example in the previous section, the expression `length**2` appeared in one of the array bound specifications. This is a particular example of a **constant expression**, which is an expression whose value cannot change during execution and which can be verified at compile time to satisfy this rule. The standard specifies a long list of possibilities for primaries in constant expressions, see below, in order that compilers can perform this verification. It is so long that if the programmer is satisfied that an expression satisfies the rule, it almost certainly does.

In the definition of a named constant we may use any constant expression, and the constant becomes defined with the value of the expression according to the rules of intrinsic assignment. This is illustrated by the example

```
integer, parameter :: length=10, long=selected_real_kind(12)
real, parameter    :: lsq = length**2
```

Note from this example that it is possible in one statement to define several named constants, in this case two, separated by commas.

A **constant expression** is an expression in which each operation is intrinsic and each primary is

i) a constant or a subobject of a constant;

ii) an array constructor in which every expression is a constant expression;

iii) a structure constructor in which each allocatable component is a reference to the intrinsic function `null`, each pointer component is an initialization target (Section 8.5.1) or a reference to `null`, and each other component is a constant expression;

iv) a specification inquiry (Section 8.18) where each designator or function argument is a constant expression or a variable whose properties inquired about are not assumed, deferred, or defined by an expression that is not a constant expression;

v) a reference to an elemental or transformational intrinsic function (Chapter 9) other than `command_argument_count`, `null`, `num_images`, `this_image`, or `transfer` provided each argument is a constant expression;

vi) a reference to the intrinsic function `null` that does not have an argument with a type parameter that is assumed or is defined by an expression that is not a constant expression;

vii) a reference to the intrinsic function `transfer` where each argument is a constant expression and each ultimate pointer component of the `source` argument is disassociated;

viii) a reference to a function of one of the intrinsic modules `ieee_arithmetic` and `ieee_exceptions` (Chapter 18), where each argument is a constant expression;

ix) within a derived-type definition, a previously declared kind type parameter of the type being defined;

x) a `do-var` within an implied-`do` loop in a data statement (Section 8.5.2);

xi) a *variable* within an array constructor (Section 7.16) where each expression of its *constructor-implied-do* is a constant expression; or

xii) a constant expression enclosed in parentheses;

where each subscript, section subscript, substring bound, or type parameter value is a constant expression.

If a constant expression invokes an inquiry function for a type parameter or an array bound of an object, the type parameter or array bound must be specified in a prior specification statement or to the left in the same specification statement.[3] Also, if in a module, a generic intrinsic function name must not be referenced if the generic name has a specific non-intrinsic version defined later in the module. Furthermore, an intrinsic function must not be referenced by its generic name if a specific non-intrinsic version is defined later.

Any named constant used in a constant expression must either be accessed from the host, be accessed from a module, be declared in a preceding statement, or be declared to the left of its use in the same statement. An example using a constant expression including a named constant that is defined in the same statement is

```
integer, parameter :: apple = 3, pear = apple**2
```

An example using one of the mathematical intrinsic functions (`sin`, `cos`, etc.) is

```
real :: root2 = sqrt(2.0)
```

8.5 Initial values for variables

8.5.1 Initialization in type declaration statements

A non-pointer non-allocatable variable may be assigned an initial value in a type declaration statement simply by following the name of the variable by an equals sign and a constant expression, as in the examples

[3] This rule is relaxed in Fortran 2018, see Section 23.3.

```
real                :: a = 0.0
real, dimension(3) :: b = [ 0.0, 1.2, 4.5 ]
```

The initial value is defined by the value of the corresponding expression according to the rules of intrinsic assignment. The variable automatically acquires the `save` attribute (Section 8.10). It must not be a dummy argument, an automatic object, or a function result.

8.5.2 The data statement

An alternative way to specify an initial value for a non-pointer non-allocatable variable is by the `data` statement. It has the general form

`data` *object-list* /*value-list*/ *[[,] object-list* /*value-list*/ *] ...*

where *object-list* is a list of variables and implied-`do` loops, and *value-list* is a list of scalar constants, structure constructors, and 'boz' constants (Section 2.6.6). A simple example is

```
real    :: a, b, c
integer :: i, j, k
data       a,b,c/1.,2.,3./, i,j,k/1,2,z'b'/
```

in which the variable `a` acquires the initial value 1.0, `b` acquires the value 2.0, ..., `k` acquires the value 11. A 'boz' constant must correspond to an integer, and this is given the value with the bit sequence of the 'boz' constant after truncation on the left or padding with zeros on the left to the bit length of the integer.

If any part of a variable is initialized by a `data` statement, the variable automatically acquires the `save` attribute. The variable must not be a dummy argument, an automatic object, or a function result.

After any array or array section in *object-list* has been expanded into a sequence of scalar elements in array element order, there must be as many constants in each *value-list* as scalar elements in the corresponding *object-list*. Each scalar element is assigned the corresponding scalar constant.

Constants which repeat may be written once and combined with a scalar integer **repeat count** which may be a literal constant, named constant, or subobject of a constant, for example

```
data i,j,k/3*0/
```

The value of the repeat count must be positive or zero. As an example, consider the statement

```
data r(1:length)/length*0./
```

where `r` is a real array and `length` is a named constant which might take the value zero.

Arrays may be initialized in three different ways: as a whole, by element, or by an implied-`do` loop. These three ways are shown below for an array declared by

```
real :: a(5, 5)
```

Firstly, for the whole array, the statement

```
data a/25*1.0/
```

sets each element of `a` to 1.0.

Secondly, individual elements and sections of `a` may be initialized, as in

```
data a(1,1), a(3,1), a(1,2), a(3,3) /2*1.0, 2*2.0/
data a(2:5,4) /4*1.0/
```

in each of which only the four specified elements and the section are initialized. Each array subscript must be a constant expression, as must any character substring subscript.

Thirdly, when the elements to be selected fall into a pattern that can be represented by `do`-loop indices, it is possible to write data statements like this:

```
data ((a(i,j), i=1,5,2), j=1,5) /15*0./
```

The general form of an implied-`do` loop is

(*dlist*, *do-var* = *expr*, *expr* [, *expr*])

where *dlist* is a list of array elements, scalar structure components, and implied-`do` loops, *do-var* is a named integer scalar variable, and each *expr* is a scalar integer expression. It is interpreted as for a `do` construct (Section 4.4), except that the `do` variable has the scope of the implied-`do` as in an array constructor (Section 7.16). A variable in an *expr* must be a *do-var* of an outer implied-`do`:

```
integer            :: j, k
integer, parameter :: lgth=5, lgth2=((lgth+1)/2)**2
real               :: a(lgth,lgth)
data ((a(j,k), k=1,j), j=1,lgth,2) / lgth2 * 1.0 /
```

This example sets to 1.0 the first element of the first row of `a`, the first three elements of the third row, and all the elements of the last row, as shown in Figure 8.1.

Figure 8.1 Result of the implied-`do` loop in the `data` statement.

1.0	.	.	.	.
.	.	.	.	.
1.0	1.0	1.0	.	.
.	.	.	.	.
1.0	1.0	1.0	1.0	1.0

The only variables permitted in subscript expressions in `data` statements are `do` indices of the same or an outer-level loop, and all operations must be intrinsic. The requirements on these expressions are identical to those on other constant expressions, allowing, for example,

```
real :: a(10,7,3)
data ((a(i,i,j),i=1,min(size(a,1),size(a,2))),j=1,size(a,3))/21*1.0/
```

An object of derived type may appear in a `data` statement. In this case, the corresponding value must be a structure constructor having a constant expression for each component. Using the derived-type definition of `posn` in Section 8.3, we can write

```
type(posn) :: position1, position2
data position1 /posn(2., 3., 0)/, position2%z /4/
```

In the examples given so far, the types and type parameters of the constants in a *value-list* have always been the same as the type of the variables in the *object-list*. This need not be the case, but they must be compatible for intrinsic assignment since the entity is initialized following the rules for intrinsic assignment. It is thus possible to write statements such as

```
data q/1/, i/3.1/, b/(0.,1.)/
```

(where `b` and `q` are real and `i` is integer).

Each variable must either have been typed in a previous type declaration statement in the scoping unit, or its type is that associated with the first letter of its name according to the implicit typing rules of the scoping unit. In the case of implicit typing, the appearance of the name of the variable in a subsequent type declaration statement in the scoping unit must confirm the type and type parameters. Similarly, any array variable must have previously been declared as such.

No variable or part of a variable may be initialized more than once in a scoping unit.

We recommend using the type declaration statement rather than the `data` statement, but the `data` statement must be employed when only part of a variable is to be initialized.

8.5.3 Pointer initialization as disassociated

Means are available to avoid the initial status of a pointer being undefined. This would be a most undesirable status since such a pointer cannot even be tested by the intrinsic function `associated` (Section 9.2). Pointers may be given the initial status of disassociated in a type declaration statement such as

```
real, pointer, dimension(:) :: vector => null()
```

a `data` statement

```
real, pointer, dimension(:) :: vector
data vector/ null() /
```

or, for procedure pointers (Section 14.2), a `procedure` statement

```
procedure(real), pointer :: pp => null()
```

This, of course, implies the `save` attribute, which applies to the pointer association status. The pointer must not be a dummy argument or function result. Here, or if the `save` attribute is undesirable (for a local variable in a recursive procedure, for example), the variable may be explicitly nullified early in the subprogram.

Our recommendation is that all pointers be so initialized to reduce the risk of bizarre effects from the accidental use of undefined pointers. This is an aid too in writing code that avoids memory leaks.

The function `null` is an intrinsic function (Section 9.17), whose simple form `null()`, as used in the above example, is almost always suitable since the attributes are immediately apparent from the context. For example, given the type `entry` of Figure 2.3, the structure constructor

```
entry (0.0, 0, null())
```

is available.

Also, pointer assignment to `null()` has the same effect as the `nullify` statement. That is, for a pointer `vector`, the statement

```
vector => null()
```

is equivalent to

```
nullify(vector)
```

The form with the argument is needed when `null` is an actual argument that corresponds to a dummy argument with assumed character length (Section 5.19) or is in a reference to a generic procedure and the type, type parameter, or rank is needed to resolve the reference (Section 5.18).

8.5.4 Pointer initialization as associated

The initial association status of a pointer can instead be defined to be associated with a target, as long as that target has the `save` attribute, is not coindexed (Section 17.4), and does not have the `allocatable` attribute. For example,

```
real, target  :: x(10,10) = 0
real, pointer :: p(:,:) => x
```

Note that the `data` statement cannot be used for this purpose; however, a procedure pointer can also be initialized to associated in a `procedure` statement, for example,

```
procedure(real), pointer :: pp => myfun
```

where `myfun` is not a dummy procedure.

Furthermore, a pointer can be associated with a part of such a target, including an array section (but not one with a vector subscript). Any subscript or substring position in the target specification must be a constant expression. For example,

```
real, pointer :: column_one(:) => x(:,1)
```

This also applies to default initialization of structure components. For example, in

```
type tpc(ipos, jpos)
   integer, kind :: ipos, jpos
   real, pointer :: pc => x(ipos, jpos)
end type
type(tpc(2, 8)) :: ps28
type(tpc(3, 5)) :: ps35
type(tpc(7, 9)) :: ps79 = tpc(x(1, 1))
```

the pointer component `ps28%pc` is associated with `x(2,8)`, `ps35%pc` is associated with `x(3,5)`, and `ps79%pc` is associated with `x(1,1)`. However, such fripperies can be a trifle confusing, so we recommend that this feature be used sparingly, and perhaps only for named pointers.

8.5.5 Default initialization of components

It is possible to specify that any object of a derived type is given a default initial value for a component. The value must be specified when the component is declared as part of the derived-type definition (Section 2.9). If the component is neither allocatable nor a pointer, this is done in the usual way (Section 8.5.1), with the equals sign followed by a constant expression, and the rules of intrinsic assignment apply (including specifying a scalar value for all the elements of an array component). If the component is a pointer, the only initialization allowed is the pointer assignment symbol followed by a reference to the intrinsic function `null` with no arguments. If the component is a allocatable, no initialization is allowed.

Initialization does not have to apply to all components of a given derived type. An example for the type defined in Figure 2.3 is

```
type entry
   real                 :: value = 2.0
   integer              :: index
   type(entry), pointer :: next => null()
end type entry
```

Given an array declaration such as

```
type(entry), dimension(100) :: matrix
```

subobjects such as `matrix(3)%value` will have the initial value 2.0, and the reference `associated(matrix(3)%next)` will return the value false.

For an object of a nested derived type, the initializations associated with components at all levels are recognized. For example, given the specifications

```
type node
   integer     :: counter
   type(entry) :: element
end type node
type (node) :: n
```

the component `n%element%value` will have the initial value 2.0.

Unlike **explicit initialization** in a type declaration or `data` statement, default initialization does not imply that the objects have the `save` attribute. However, all data objects declared in a module do have the `save` attribute.

Objects may still be explicitly initialized in a type declaration statement, as in

```
type(entry), dimension(100) :: matrix=entry(huge(0.0),huge(0),null())
```

in which case the default initialization is ignored. Similarly, default initialization may be overridden in a nested derived-type definition such as

```
type node
   integer     :: counter
   type(entry) :: element=entry(0.0, 0 , null())
end type node
```

However, no part of a non-pointer object with default initialization is permitted in a `data` statement (Section 8.5.2).

As well as applying to the initial values of static data, default initialization also applies to any data that is dynamically created during program execution. This includes allocation with the `allocate` statement. For example, the statement

```
allocate (matrix(1)%next)
```

creates a partially initialized object of type `entry`. It also applies to unsaved local variables (including automatic objects), function results, and dummy arguments with intent `out`.

It applies even if the derived-type definition is `private` (Section 8.6.1) or the components are `private`.

8.6 Accessibility

8.6.1 The public and private attributes

Modules (Section 5.5) permit specifications to be 'packaged' into a form that allows them to be accessed elsewhere in the program. So far, we have assumed that all the entities in the module are to be accessible, that is have the `public` attribute, but sometimes it is desirable to limit the access. For example, several procedures in a module may need access to a work array containing the results of calculations that they have performed. If access is limited to only the procedures of the module, there is no possibility of an accidental corruption of these data by another procedure and design changes can be made within the module without affecting the rest of the program. In cases where entities are not to be accessible outside their own module, they may be given the `private` attribute.

These two attributes may be specified with the `public` and `private` attributes on type declaration statements in the module, as in

```
real, public     :: x, y, z
integer, private :: u, v, w
```

or in `public` and `private` statements, as in

```
public  :: x, y, z, operator(.add.)
private :: u, v, w, assignment(=), operator(*)
```

which have the general forms

`public` *[[::] access-id-list]*
`private` *[[::] access-id-list]*

where *access-id* is a name or a *generic-spec* (Section 5.18).

Note that if a procedure has such a generic identifier, the accessibility of its specific name is independent of the accessibility of its generic identifier. One may be `public` while the other is `private`, which means that it is accessible only by its specific name or only by its generic identifier.

If a `public` or `private` statement has no list of entities, it confirms or resets the default. Thus, the statement

```
public
```

confirms `public` as the default value, and the statement

```
private
```

sets the default value for the module to `private` accessibility. For example,

```
private
public :: means
```

gives the entity `means` the `public` attribute whilst all others are `private`. There may be at most one accessibility statement without a list in a scoping unit.

The entities that may be specified by name in `public` or `private` lists are named variables, procedures (including generic procedures), derived types, named constants, and namelist groups.[4] Thus, to make a generic procedure name accessible but the corresponding specific names inaccessible, we might write the code of Figure 8.2.

Figure 8.2 Making a generic procedure name accessible and the specific names inaccessible.

```
module example
   private specific_int, specific_real
   interface generic_name
      module procedure specific_int, specific_real
   end interface
contains
   subroutine specific_int(i)
      ⋮
   subroutine specific_real(a)
      ⋮
end module example
```

A type that is accessed from a module may be given the `private` attribute in the accessing module. If an entity of this type has the `public` attribute, a subsequent `use` statement for it may be accompanied by a `use` statement for the type from the original module.

Entities of private type are not themselves required to be private; this applies equally to procedures with arguments that have private type. This means that a module writer can provide very limited access to values or variables without thereby giving the user the power to create new variables of the type. For example, the widely used LAPACK library requires character arguments such as `uplo`, a character variable that must be given the value `'L'` or `'U'` according to whether the matrix is upper or lower triangular. The value is checked at run time and an error return occurs if it is invalid. This could be replaced by values `lower` and `upper` of private type. This would be clearer and the check would be made at compile time.

The use of the `private` statement for components of derived types in the context of defining an entity's access within a module will be described in Section 8.15.

The `public` and `private` attributes may appear only in the specifications of a module.

[4] In Fortran 2018, module names are also permitted, see Section 23.1. This sets the default accessibility for entities accessed from the module.

8.6.2 More control of access from a module

It is sometimes desirable to allow the user of a module to be able to reference the value of a module variable without allowing it to be changed. Such control is provided by the `protected` attribute. This attribute does not affect the visibility of the variable, which must still be `public` to be visible, but confers the same protection against modification that intent `in` does for dummy arguments. It may appear only in the specifications of a module.

The `protected` attribute may be specified with the `protected` keyword in a type declaration statement. For example, in

```
module m
   public
   real, protected    :: v
   integer, protected :: i
```

both `v` and `i` have the `protected` attribute. The attribute may also be specified separately, in a `protected` statement, just as for other attributes (see Section 8.8).

Variables with this attribute may only be modified within the defining module. Outside the module they are not allowed to appear in a context in which they would be altered, such as on the left-hand side of an assignment statement.

For example, in the code of Figure 8.3, the `protected` attribute allows users of `thermometer` to read the temperature in either Fahrenheit or Celsius, but the variables can only be changed via the provided subroutines which ensure that both values agree.

Figure 8.3 Using `protected` to ensure the consistency of Fahrenheit and Celsius values.

```
module thermometer
  real, protected :: temperature_celsius   = 0
  real, protected :: temperature_fahrenheit = 32
contains
  subroutine set_celsius(new_celsius_value)
    real, intent(in) :: new_celsius_value
    temperature_celsius = new_celsius_value
    temperature_fahrenheit = temperature_celsius*(9.0/5.0) + 32
  end subroutine set_celsius
  subroutine set_fahrenheit(new_fahrenheit_value)
    real, intent(in) :: new_fahrenheit_value
    temperature_fahrenheit = new_fahrenheit_value
    temperature_celsius = (temperature_fahrenheit - 32)*(5.0/9.0)
  end subroutine set_fahrenheit
end module thermometer
```

8.7 Pointer functions denoting variables

When a **pointer function** returns an associated pointer, that pointer is always associated with a variable that has the `target` attribute, either by pointer assignment or by allocation. Such

a reference to a pointer function may also be used in contexts that require a variable, in particular

- as an actual argument for an intent `inout` or `out` dummy argument;
- on the left-hand side of an assignment statement.

In this respect, a pointer function reference can be used exactly as if it were the variable that is the target of the pointer result.

These are sometimes known as **accessor functions**; by abstracting the location of the variable, they enable objects with special features such as sparse storage, instrumented accesses, and so on to be used as if they were normal arrays. They also allow changing the underlying implementation mechanisms without needing to change the code using the objects. An example of this feature is shown in Figure 8.4.

Figure 8.4 Example of accessor functions. The function `findloc` is defined in Section 9.16.

```
module indexed_store
  real, private, pointer    :: values(:) => null()
  integer, private, pointer :: keys(:) => null()
  integer, private          :: maxvals = 0
contains
  function storage(key)
    integer, intent(in) :: key
    real, pointer :: storage
    integer :: loc
    if (.not.associated(values)) then
      allocate (values(100), keys(100))
      keys(1) = key
      storage => values(1)
      maxvals = 1
    else
      loc = findloc(keys(:maxvals), key)
      if (loc>0) then
        storage => values(loc)
      else
        ⋮  (Code to store new element elided.)
      end if
    end if
  end function
end module
⋮
storage(13) = 100
print *, storage(13)
```

8.8 The pointer, target, and allocatable statements

For the sake of regularity in the language, there are statements for specifying the `allocatable`, `pointer`, and `target` attributes of entities. They take the forms:

`allocatable` *[*`::`*] object-decl-list*
`pointer` *[*`::`*] pointer-decl-list*
`target` *[*`::`*] object-decl-list*

where *pointer-decl* is an *object-decl* or *proc-name* and each *object-decl* has the form

object-name [`(`*array-spec*`)`*] [*`[`*coarray-spec*`]`*]*

Each *array-spec* provides rank and bounds information for an array, and each *coarray-spec* provides corank and cobounds information for a coarray (Chapter 17). They must satisfy the same rules as apply to providing this information in a type declaration statement for an array or coarray with the `allocatable`, `pointer`, or `target` attribute, respectively.

A *proc-name* in a `pointer` statement specifies that the procedute name is a procedure pointer.

If the array information for an object is not supplied, this may be because the object is scalar or it may be because this information is supplied on another statement. Similarly, if the coarray information is not supplied, this may be because the object is not a coarray or it may be because this information is supplied on another statement. Neither the attribute, the array information, nor the coarray imformation may be supplied twice for an entity.

Here is an example of the use of this feature:

```
real        :: a, son, y
allocatable :: a(:,:)
pointer     :: son
target      :: a, y(10)
```

We believe that it is much clearer to specify these attributes for objects on type declaration statements, and therefore do not use these forms for objects. However, if a procedure pointer is declared with an interface block, a separate `pointer` statement is needed to give it the pointer attribute.

8.9 The intent and optional statements

The `intent` attribute (Section 5.9) for a dummy argument that is not a dummy procedure (but may be a dummy procedure pointer) may be specified in a type declaration statement, a `procedure` statement, or in an `intent` statement of the form

`intent(`*inout*`)` *[*`::`*] dummy-argument-name-list*

where *inout* is `in`, `out`, or `inout`. Examples are

```
subroutine solve (a, b, c, x, y, z)
   real        :: a, b, c, x, y, z
   intent(in)  :: a, b, c
   intent(out) :: x, y, z
```

The `optional` attribute (Section 5.13) for a dummy argument may be specified in a type declaration statement or in an `optional` statement of the form

```
optional [::] dummy-argument-name-list
```

An example is

```
optional :: a, b, c
```

The `optional` attribute is the only attribute which may be specified for a dummy argument that is a procedure.

Note that the `intent` and `optional` attributes may be specified only for dummy arguments. As for the statements of Section 8.8, we believe that it is much clearer to specify these attributes for objects on the type declaration statements, and therefore do not use these forms for objects. However, if a procedure pointer is declared with an interface block, a separate `optional` statement is needed to give it the optional attribute.

8.10 The save attribute

Let us suppose that we wish to retain the value of a local variable in a subprogram, for example to count the number of times the subprogram is entered. We might write a section of code as in Figure 8.5. In this example, the local variable `counter` is initialized to zero and it is assumed that the current values of `a` and `counter` are available each time the subroutine is called. This is not necessarily the case. Fortran allows the computer system being used to 'forget' a new value, the variable becoming undefined on each return unless it has the `save` attribute. In Figure 8.5, it is sufficient to change the declaration of `a` to

```
real, save :: a
```

to be sure that its value is always retained between calls. This may be done for `counter`, too, but is not necessary as all variables with initial values (Section 8.5.1) acquire the `save` attribute automatically.

Figure 8.5 Counting the number of times a procedure is invoked.

```
subroutine anything(x)
   real    :: a, x
   integer :: counter = 0 ! Initialize the counter
   ⋮
   counter = counter + 1
   if (counter==1) then
      a = x
   else
      a = a + x
   end if
```

A similar situation cannot arise with the use of variables in modules (Section 5.5) because all data objects in a module automatically have the `save` attribute.

If a variable that becomes undefined on return from a subprogram has a pointer associated with it, the pointer's association status becomes undefined.

The `save` attribute must not be specified for a dummy argument, a function result, or an automatic object (Section 7.3). It may be specified for a pointer, in which case the pointer association status is saved. It may be specified for an allocatable array, in which case the allocation status and value are saved. A saved variable in a subprogram that is referenced recursively is shared by all executing instances of the subprogram.

An alternative to specifying the `save` attribute on a type declaration statement is the `save` statement:

`save` *[[::] variable-name-list]*

A `save` statement with no list is equivalent to a list containing all possible names, and in this case the scoping unit must contain no other `save` statements and no `save` attributes in type declaration statements. Our recommendation is against this form of `save`. If a programmer tries to give the `save` attribute explicitly to an automatic object, a diagnostic will result. On the other hand, he or she might think that `save` without a list would do this too, and not get the behaviour intended. Also, there is a loss of efficiency associated with `save` on some processors, so it is best to restrict it to those objects for which it is really needed.

The `save` statement or `save` attribute may appear in the declaration statements in a main program but has no effect.

8.11 Volatility

8.11.1 The volatile attribute

The `volatile` attribute may be applied only to variables and is conferred either by the `volatile` attribute in a type declaration statement, or by the `volatile` statement, which has the form

`volatile` *[::] variable-name-list*

For example,

```
integer, volatile :: x
real              :: y
volatile          :: y
```

declares two volatile variables `x` and `y`.

Being volatile indicates to the compiler that, at any time, the variable might be changed and/or examined from outside the Fortran program. This means that each reference to the variable will need to load its value from main memory (so, for example, it cannot be kept in a register in an inner loop). Similarly, each assignment to the variable must write the data to memory. Essentially, this disables most optimizations that might have been applicable to the object, making the program run slower but, one hopes, making it work with some special hardware or multi-processing software.

However, it is the responsibility of the programmer to effect any necessary synchronization; this is particularly relevant to multi-processor systems. Even if only one process is writing to the variable and the Fortran program is reading from it, because the variable is not automatically protected by a critical section (see Section 17.13.4) it is possible to read a partially updated (and thus an inconsistent or impossible) value. For example, if the variable

is an IEEE floating-point variable, reading a partially updated value could return a signalling NaN; or if the variable is a pointer, its descriptor might be invalid. In either of these cases the program could be abruptly terminated, so this facility must be used with care.

Similarly, if two processes are both attempting to update a single `volatile` variable, the effects are completely processor dependent. The variable might end up with its original value, one of the values from an updating process, a partial conglomeration of values from the updating processes, or the program could even crash.

A simple use of this feature might be to handle some external (interrupt-driven) event, such as the user typing Control-C, in a controlled fashion. In the Figure 8.6 example, `register_event_flag` is a subroutine, possibly written in another language, which ensures that `event_has_occurred` becomes true when the specified event occurs.

Figure 8.6 Handling an external event.

```
logical, target, volatile :: event_has_occurred
⋮
event_has_occurred = .false.
call register_event_flag(event_has_occurred, ...)
⋮
do
  ⋮                              ! some calculations
  if (event_has_occurred) exit ! exit loop if event happened
  ⋮                              ! some more calculations
  if (...) exit                  ! Finished our calculations yet?
end do
⋮
```

If the variable is a pointer, the `volatile` attribute applies both to the descriptor and to the target. Even if the target does not have the `volatile` attribute, it is treated as having it when accessed via a pointer that has it. If the variable is allocatable, it applies both to the allocation and to the value. In both cases, if the variable is polymorphic (Section 15.3), the dynamic type may change by non-Fortran means.

If a variable has the `volatile` attribute, so do all of its subobjects. For example, in

```
logical, target            :: signal_state(100)
logical, pointer, volatile :: signal_flags(:)
⋮
signal_flags => signal_state
⋮
signal_flags(10) = .true.    ! A volatile reference
⋮
write (20) signal_state      ! A nonvolatile reference
```

the pointer (descriptor) of `signal_flags` is volatile, and access to each element of `signal_flags` is volatile; however, `signal_state` itself is not volatile.

The raison d'être for `volatile` is for interoperating with parallel-processing packages such as MPI, which have procedures for asynchronously transferring data from one process to another. For example, without the `volatile` attribute on the array `data` in Figure 8.7, a compiler optimization could move the assignment prior to the call to `mpi_wait`. The use of `mpi_module` provides access to MPI constants and explicit interfaces for MPI functions, in particular `mpi_isend`, which requires the `volatile` attribute on its first dummy argument.

Figure 8.7 Using `volatile` to avoid code motion.

```
subroutine transfer_while_producing(...)
   use mpi_module ! Access interfaces for mpi_isend etc.
   real, allocatable           :: newdata(:)
   real, allocatable, volatile :: data(:)
   ⋮
   ⋮ ! Produce data here
   call mpi_isend(data, size(data), mpi_real, dest, &
                  tag, comm, request, ierr)
   ⋮
   ⋮ ! Produce newdata here
   call mpi_wait(request, status)
   data = newdata
   ⋮
end subroutine transfer_while_producing
```

8.11.2 Volatile scoping

If a variable only needs to be treated as volatile for a short time, the programmer has two options: either to pass it to a procedure to be acted on in a volatile manner (see Section 8.11.3), or to access it by use or host association, using a `volatile` statement to declare it to be volatile only in the accessing scope. For example, in the code of Figure 8.8, the `data` array is not volatile in `data_processing`, but is in `data_transfer`. Note that this is an exception to the usual rules of use association, which prohibit other attributes from being changed in the accessing scope. Similarly, declaring a variable that is accessed by host association to be volatile is allowed, and unlike other specification statements, does not cause the creation of a new local variable.

8.11.3 Volatile arguments

The volatility of an actual argument and its associated dummy argument may differ. This is important since volatility may be needed in one but not in the other. In particular, a volatile variable may be used as an actual argument in a call to an intrinsic procedure. However, while a volatile variable is associated with a non-volatile dummy argument, the programmer must ensure that the value is not altered by non-Fortran means. Note that, if the volatility of

Figure 8.8 Using a procedure to limit the scope of a variable's volatility.

```
module data_module
   real, allocatable :: data(:,:), newdata(:,:)
   ⋮
contains
   subroutine data_processing
      ⋮
   end subroutine data_processing
   subroutine data_transfer
      volatile :: data
      ⋮
   end subroutine data_transfer
end module data_module
```

an actual argument persists through a procedure reference, as for example in the MPI call in Figure 8.7, this means that the procedure referenced must have an explicit interface and the corresponding dummy argument must be declared to be volatile.

If the procedure is non-elemental and the dummy argument is a volatile array, the actual argument must not be an array section with a vector subscript; furthermore, if the actual argument is an array section or an assumed-shape array, the dummy argument must be assumed-shape and if the actual argument is an array pointer, the dummy argument must be a pointer or assumed-shape. These restrictions are designed to allow the argument to be passed by reference; in particular, to avoid the need for a local copy being made as this would interfere with the volatility.

A dummy argument with intent `in` or the `value` attribute (Section 19.8) is not permitted to be volatile. This is because the value of such an argument is expected to remain fixed during the execution of the procedure.

If a dummy argument of a procedure is volatile, the interface must be explicit whenever it is called and the dummy argument must be declared as volatile in any interface body for the procedure.

8.12 The asynchronous attribute

The `asynchronous` attribute for a variable concerns solely input/output operations, and its description is therefore deferred to Section 10.15.

8.13 The block construct

The `block` construct is a scoping unit that is an executable construct, providing the ability to declare entities within the executable part of a subprogram that have the scope of the construct. Such entities may be variables, types, constants, or even external procedures. Each

is known as a **construct entity**. Any entity of the host scoping unit with the same name is hidden by the declaration.

For example, in

```
do i=1, m
  block
    real alpha, temp(n)
    integer j
    ⋮
    temp(j) = alpha*a(j, i) + b(j)
    ⋮
  end block
end do
```

the variables `alpha`, `temp`, and `j` are local to the block, and have no effect on any variables outside the block that might have the same names. Used judiciously, this can make code easier to understand (there is no need to look through the whole subprogram for later accesses to `alpha`, for instance) and since the compiler also knows that these are local to each iteration, this can aid optimization. Adding a block construct to existing code has no effect on the semantics of the existing code.

Another example is

```
block
  use convolution_module
  intrinsic norm2
  ⋮
  x = convolute(y)*norm2(z)
  ⋮
end block
```

Here, the entities brought in by the `use` statement are visible only within the block, and the declaration of the `norm2` intrinsic avoids clashing with any `norm2` that might exist outside the block. These techniques can be useful in large subprograms, or during code maintenance when it is desired to access a module or procedure without risking disturbance to the rest of the subprogram.

Not all declarations are permitted in a `block` construct. The `intent`, `optional`, and `value` attributes are not available (because a `block` has no dummy arguments), and the `implicit` statement is prohibited because it would be confusing to change the implicit typing rules in the middle of a subprogram. The `namelist` statement (Section 8.20) is prohibited because of potential ambiguity or confusion. Finally, a `save` statement that specifies entities in the `block` is permitted, but a global `save` is prohibited, again because it would be ambiguous as to just exactly what would be saved.

Like other constructs, the `block` construct may be given a construct name, and that construct name may be used in `exit` statements to exit from the construct (see Section 4.5). Similarly, `block` constructs may be nested the same way that other constructs are nested. An example of this is shown in Figure 8.9.

Figure 8.9 Nesting `block` constructs. For `epsilon`, see Section 9.9.2.

```
find_solution: block
  real :: work(n)
  ⋮
  loop: do i=1, n
     block
        real :: residual
        ⋮
        if (residual < epsilon(x)) exit find_solution
      end block
   end do loop
   ⋮
 end block find_solution
```

The `block` construct is only of limited use in normal programming, but is really useful when program-generation techniques such as macros are being used, to avoid conflicts with entities elsewhere in a subprogram. (Macros are not part of Fortran, but various macro processors are widely used with Fortran.)

8.14 The use statement

In Section 5.5 we introduced the `use` statement in its simplest form

`use` *module-name*

which provides access to all the public named data objects, derived types, interface blocks, procedures, generic identifiers, and namelist groups (Section 8.20) in the module named. Any `use` statements must precede other specification statements in a scoping unit. Apart from `volatile`, the only attribute of an accessed entity that may be specified afresh is `public` or `private` (and this only in a module), but the entity may be included in one or more namelist groups.

If access is needed to two or more modules that have been written independently, the same name might be in use in more than one module. This is the main reason for permitting accessed entities to be renamed by the `use` statement. Renaming is also available to resolve a name clash between a local entity and an entity accessed from a module, though our preference is to use a text editor or other tool to change the local name. With renaming, the `use` statement has the form

`use` *module-name*`,` *rename-list*

where each *rename* has the form

local-id `=>` *use-id*

where *use-id* is a public name or *generic-spec* (for a user-defined operator) in the module, and *local-id* is the identifier to use for it locally.

As an example,

```
use stats_lib, sprod => prod
use maths_lib
```

makes all the public entities in both `stats_lib` and `maths_lib` accessible. If `maths_lib` contains an entity called `prod`, it is accessible by its own name while the entity `prod` of `stats_lib` is accessible as `sprod`.

Renaming is not needed if there is a name clash between two entities that are not required. A name clash is permitted if there is no reference to the name in the scoping unit.

A name clash is also permissible for a generic name that is required. Here, all generic interfaces accessed by the name are treated as a single concatenated interface block. This is true too for defined operators, and also for defined assignment (for which no renaming facility is available). In all these cases, any two procedures having the same generic identifier must differ as explained in Section 5.18. We imagine that this will usually be exactly what is needed. For example, we might access modules for interval arithmetic and matrix arithmetic, both needing the functions `sqrt`, `sin`, etc., the operators `+`, `-`, etc., and assignment, but for different types.

Just as with variable and procedure names, also user-defined operators may be renamed on a `use` statement. For example,

```
use fred, operator(.nurke.) => operator(.banana.)
```

renames the `.banana.` operator located in module `fred` so that it may be referenced by using `.nurke.` as an operator.

However, this only applies to user-defined operators. Intrinsic operators cannot be renamed, so all of the following are invalid:

```
use fred, only: operator(.equal.) => operator(.eq.)     ! Invalid
use fred, only: operator(.ne.) => operator(.notequal.) ! Invalid
use fred, only: operator(*) => assignment(=)           ! Invalid
```

For cases where only a subset of the names of a module is needed, the `only` option is available, having the form

`use` *module-name*`, only:` *[only-list]*

where each *only* has the form

access-id

or

[local-id =>] use-id

where each *access-id* is a public entity in the module, and is either a name or a *generic-spec* (Section 5.18). This provides access to an entity in a module only if the entity is public and is specified as a *use-id* or *access-id*. Where a *use-id* is preceded by a *local-id*, the entity is known locally by the *local-id*. An example of such a statement is

```
use stats_lib, only : sprod => prod, mult
```

which provides access to `prod` by the local name `sprod` and to `mult` by its own name.

We would recommend that only one `use` statement for a given module be placed in a scoping unit, but more are allowed. If there is a `use` statement without an `only` qualifier, all

public entities in the module are accessible and the *rename-list*s and *only-list*s are interpreted as if concatenated into a single *rename-list* (with the form *use-id* in an *only-list* being treated as the rename *use-id* => *use-id*). If all the statements have the `only` qualification, only those entities named in one or more of the *only-list*s are accessible, that is all the *only-list*s are interpreted as if concatenated into a single *only-list*.

An `only` list will be rather clumsy if almost all of a module is wanted. The effect of an 'except' clause can be obtained by renaming unwanted entities. For example, if a large program (such as one written in Fortran 77) contains many external procedures, a good practice is to collect interface blocks for them all into a module that is referenced in each program unit for complete mutual checking. In an external procedure, we might then write:

```
use all_interfaces, except_this_one => name
```

to avoid having two explicit interfaces for itself (where `all_interfaces` is the module name and `name` is the procedure name).

When a module contains `use` statements, the entities accessed are treated as entities in the module. They may be given the `private` or `public` attribute explicitly or through the default rule in effect in the module. Thus, given the two modules in Figure 8.10 and a third program unit containg a `use` statement for `two`, the variable `i` is accessible there only if it also contains a `use` statement for `one` or if `i` is made public explicitly in `two`.

Figure 8.10 Making private an entity accessed from a module.

```
module one
   integer :: i
end module one
module two
   use one
   private
   ⋮
end module two
```

An entity may be accessed by more than one local name. This is illustrated in Figure 8.11, where module `b` accesses `s` of module `a` by the local name `bs`; if a subprogram such as `c` accesses both `a` and `b`, it will access `s` by both its original name and by the name `bs`. Figure 8.11 also illustrates that an entity may be accessed by the same name by more than one route (see variable `t`).

A more direct way for an entity to be accessed by more than one local name is for it to appear more than once as a *use-name*. This is not a practice that we recommend.

Of course, all the local names of entities accessed from modules must differ from each other and from names of local entities. If a local entity is accidentally given the same name as an accessible entity from a module, this will be noticed at compile time if the local entity is declared explicitly (since no accessed entity may be given any attribute locally, other than `private` or `public`, and that only in a module). However, if the local entity is intended to be implicitly typed (Section 8.2) and appears in no specification statements, each appearance of the name will be taken, incorrectly, as a reference to the accessed variable. To avoid this, we

Figure 8.11 Accessing a variable by more than one local name.

```
module a
   real :: s, t
   ⋮
end module a
module b
   use a, bs => s
   ⋮
end module b
subroutine c
   use a
   use b
   ⋮
end subroutine c
```

recommend, as always, the conscientious use of explicit typing in a scoping unit containing one or more `use` statements. For greater safety, the `only` option may be employed on a `use` statement to ensure that all accesses are intentional.

8.15 Derived-type definitions

When derived types were introduced in Section 2.9, some simple example definitions were given, but the full generality was not included. An example illustrating more features is

```
type, public :: lock
   private
   integer, pointer :: key(:)
   logical          :: state
end type lock
```

The general form (apart from redundant features, see Appendix A.2) is

`type` *[[,* *type-attr* *] ...* `::` *]* *type-name* *[* `(` *type-param-name-list* `)` *]*
[*type-param-def-stmt* *] ...*
[`private` *]*
[*component-def-stmt* *] ...*
[*type-bound-procedure-part* *]*
`end type` *[* *type-name* *]*

where each *type-attr* is one of

`abstract`	(Section 15.10),
`access`	(described later in this section),
`bind(c)`	(Section 19.4), or
`extends(`*parent-type-name*`)`	(Section 15.2).

Derived type parameters (*type-param-name-list* and *type-param-def-stmt*) are described in Section 13.2. The *type-bound-procedure-part* is described in Section 15.8.

Each *component-def-stmt* is either a `procedure` statement (Section 14.2.2) or declares one or more data components and has the form

type [[, component-attr] ... ::] component-decl-list

where *type* specifies the type and type parameters (see Section 8.17), each *component-attr* is *access*, `allocatable` (Section 6.11), `codimension[`*cobounds-list*`]` (Section 17.8), `contiguous` (Section 7.18.1), `dimension(`*bounds-list*`)`, or `pointer`, and each *component-decl* is

*component-name [(bounds-list)] [*char-len] [[cobounds-list]] [comp-init]*

The meaning of **char-len* is explained at the end of Section 8.17 and *comp-int* represents component initialization, as explained in Section 8.5.5. If the *type* is a derived type and neither the `allocatable` nor the `pointer` attribute is specified, the type must be previously defined in the host scoping unit or accessible there by use or host association. If the `allocatable` or `pointer` attribute is specified, the type may also be the one being defined (for example, the type `entry` of Section 2.3), or one defined elsewhere in the scoping unit.

A *type-name* must not be the same as the name of any intrinsic type or a derived type accessed from a module.

The bounds of an array component are declared by a *bounds-list*, where each *bounds* is just

:

for an allocatable or a pointer component (see example in Section 7.13) or

[lower-bound:] upper-bound

for a component that is neither allocatable nor a pointer and *lower-bound* and *upper-bound* are specification expressions (Section 8.18) whose values do not depend on those of variables. Similarly, the character length of a component of type character must be a specification expression whose value does not depend on that of a variable. If there is a *bounds-list* attached to the *component-name*, this defines the bounds. If a `dimension` attribute is present in the statement, its *bounds-list* applies to any component in the statement without its own *bounds-list*. Similarly, a `codimension` attribute in the statement applies to any component in the statement without its own *cobounds-list*.

Only if the host scoping unit is a module may the `private` statement, or an *access* attribute on the `type` statement, or a *component-def-stmt* appear. The *access* attribute on a type statement may be `public` or `private` and specifies the accessibility of the type. If it is `private`, then the type name, the structure constructor for the type, any entity of the type, and any procedure with a dummy argument or function result of the type are all inaccessible outside the host module. The accessibility may also be specified in a `private` or `public` statement in the host. In the absence of both of these, the type takes the default accessibility of the host module.

If a `private` statement appears, any component whose *component-def-stmt* does not have a `public` attribute is private. A component that is private (whether by a `private` statement or `private` attribute) is inaccessible in any scoping unit accessing the host module; this means

it cannot be selected as a component, and a structure constructor cannot provide an explicit value for it.

The accessibility of a type and of each component are completely independent, so all combinations are possible: a public component in a public type, a public component in a private type, a private component in a public type, and a private component in a private type. Even when the type and all of its components are private, entities of that type can be accessible: that is, public variables, constants, functions, and components of other types can be of the private type.

Having mixed component accessibility can be useful in some situations. For example, in

```
module mytype_module
   type mytype
      private
      character(20), public :: debug_tag = ''
      ⋮     ! private components omitted
   end type mytype
   ⋮
end module mytype_module
```

although some of the components of `mytype` are private (protecting the integrity of those components), the `debug_tag` field is public so it can be used directly outside of the module `mytype_module`. If any component of a derived type is private, the structure constructor can be used outside the module in which it is defined only if that component has default initialization, which allows the value for that component to be omitted.

In practice the full generality of accessibility is often not needed, most cases being effectively addressed by one of three simpler levels of access:

i) all `public`, where the type and all its components are accessible, and the components of any object of the type are accessible wherever the object is accessible;

ii) a `public` type with all `private` components, where the type is accessible but all of its components are hidden (commonly referred to as an opaque type);

iii) all `private`, where both the type and its components are used only within the host module, and are hidden to an accessing procedure.

An opaque type (case ii) has, where appropriate, the advantage of enabling changes to be made to the type without in any way affecting the code in the accessing procedure. A fully private type (case iii) offers this advantage and has the additional merit of not cluttering the name space of the accessing procedure. The use of `private` accessibility for the components or for the whole type is thus recommended whenever possible.

We note that, even if two derived-type definitions are identical in every respect except their names, then entities of those two types are not equivalent and are regarded as being of different types. Even if the names, too, are identical, the types are different (unless they have the `sequence` attribute, a feature that we do not recommend and whose description is left to Appendix A.2). If a type is needed in more than one program unit, the definition should be placed in a module and accessed by a `use` statement wherever it is needed. Having a single definition is far less prone to errors.

8.16 The type declaration statement

We have already met many simple examples of the declarations of named entities by `integer`, `real`, `complex`, `logical`, `character`, and `type`(*type-name*) statements. The general form is

type [[, type-attr] ... ::] entity-list

where *type* specifies the type and type parameters (Section 8.17), *type-attr* is one of the following:

`allocatable`	`dimension(`*bounds-list*`)`	`parameter`	`save`
`asynchronous`	`external`	`pointer`	`target`
`bind(c...)`	`intent(`*inout*`)`	`private`	`value`
`codimension[`*cobounds-list*`]`	`intrinsic`	`protected`	`volatile`
`contiguous`	`optional`	`public`	

and each *entity* is one of

*object-name [(bounds-list)] [[cobounds-list]] [*char-len] [= constant-expr]*
*function-name [*char-len]*
*pointer-name [(bounds-list)] [*char-len] [=> null-init]*

where *null-init* is a reference to the intrinsic function `null` with no arguments. The meaning of **char-len* is explained at the end of Section 8.17; a *bounds-list* specifies the rank and possibly bounds of array-valued entities. If =*constant-expr* or =>*null-init* appears, any array bound or character length that is an expression must be a constant expression.

No attribute may appear more than once in a given type declaration statement. The double colon `::` need not appear in the simple case without any *attribute*s and without any initialization (= *constant-expr* or => *null-init*); for example

```
real a, b, c(10)
```

If the statement specifies the `parameter` attribute, = *constant-expr* must appear.

If the `pointer` attribute is specified, the `allocatable`, `intrinsic`, and `target` attributes must not be specified. If the `parameter` attribute is specified, the `allocatable`, `external`, `intent`, `intrinsic`, `save`, `optional`, and `target` attributes must not be specified. If either the `external` or the `intrinsic` attribute is specified, the `target` attribute must not be specified.

If an object is specified with the `intent` or `parameter` attribute, this is shared by all its subobjects. The `pointer` attribute is not shared in this manner, but note that a derived-data type component may itself be a pointer. However, the `target` attribute is shared by all its subobjects, except for any that are pointer components.

The `bind`, `intrinsic`, `parameter`, and `save` attributes must not be specified for a dummy argument or function result.

The `intent`, `optional`, and `value` attributes may be specified only for dummy arguments.

For a function result, specifying the `external` attribute is an alternative to the external statement (Section 5.11) for declaring the function to be external, and specifying the

`intrinsic` attribute is an alternative to the intrinsic statement (Section 9.1.3) for declaring the function to be intrinsic. These two attributes are mutually exclusive.

Each of the attributes may also be specified in statements (such as `save`) that list entities having the attribute. This leads to the possibility of an attribute being specified explicitly more than once for a given entity, but this is not permitted. Our recommendation is to avoid such statements because it is much clearer to have all the attributes for an entity collected in one place.

8.17 Type and type parameter specification

We have used *type* to represent one of the following

```
integer    [ ( [ kind= ] kind-value ) ]
real       [ ( [ kind= ] kind-value ) ]
complex    [ ( [ kind= ] kind-value ) ]
character  [ ( actual-parameter-list ) ]
logical    [ ( [ kind= ] kind-value ) ]
type       ( type-name [ actual-parameter-list ] )
```

in the `function` statement (Section 5.20), the component definition statement (Section 8.15), and the type declaration statement (Section 8.16). A *kind-value* must be a constant expression (Section 8.4) and must have a value that is valid on the processor being used.

For `character`, each *actual-parameter* has the form

[`len=` *]* *len-value* or *[* `kind=` *]* *kind-value*

and provides a value for one of the parameters. It is permissible to omit `kind=` from a kind *actual-parameter* only when `len=` is omitted and *len-value* is both present and comes first, just as for an actual argument list (Section 5.13). Neither parameter may be specified more than once. The *actual-parameter-list* for a derived type is described in Section 13.2.

For a scalar named constant or for a dummy argument of a subprogram, a *len-value* may be specified as an asterisk, in which case the value is assumed from that of the constant itself or the associated actual argument. In both cases, the `len` intrinsic function (Section 9.7.1) is available if the actual length is required directly, for instance as a `do`-construct iteration count. A combined example is

```
character(len=len(char_arg)) function line(char_arg)
   character(len=*)             :: char_arg
   character(len=*), parameter :: char_const = 'page'
   if ( len(char_arg) < len(char_const) ) then
   ⋮
```

A *len-value* that is neither an asterisk nor, later, a colon (see Section 6.3) must be a specification expression (Section 8.18) and must be a constant expression if *=constant-expr* or *=>null-init* appears. Negative values declare character entities to be of zero length.

In addition, it is possible to specify the character length for individual entities in a type declaration statement or component declaration statement using the syntax *entity*char-len*, where *char-len* is either (*len-value*) or *len*, where *len* is a scalar integer literal constant that does not have a kind type parameter specified for it. An illustration of this form is

```
character(len=8) :: word(4), point*1, text(20)*4
```

where, `word`, `point`, and `text` have character length 8, 1, and 4, respectively. Similarly, the alternative form may be used for individual components in a component definition statement.

8.18 Specification expressions

A scalar integer expression known as a **specification expression** may be used to specify the array bounds (examples in Section 7.3) and character lengths of data objects in a subprogram, and of function results. Such an expression need not be constant but may depend only on data values that are always available on entry to the subprogram. Just as for constant expressions (Section 8.4), the standard specifies a long list of rules in order to ensure that this is so and that the execution of a specification expression does not have undesirable side effects.

Non-intrinsic functions are permitted to be called. They are required to be pure to ensure that they cannot have side effects on other objects being declared in the same specification sequence. They are required not to be internal, which ensures that they cannot inquire, via host association, about other objects being declared. Recursion is disallowed, to avoid the creation of a new instance of a procedure while the construction of one is in progress.

We begin by defining a **specification inquiry**, which is a a reference to

i) an intrinsic inquiry function (Section 9.1.2), other than `present`,

ii) a type parameter inquiry (Section 13.1),

iii) an inquiry function from the intrinsic module `ieee_arithmetic` or `ieee_exceptions`,

iv) the function `c_sizeof` from the intrinsic module `iso_c_binding`, or

v) the function `compiler_version` or `compiler_options` from the intrinsic module `iso_fortran_env`.

Next, a function is a **specification function** if it is a pure function, is not a standard intrinsic function, is not an internal function, and does not have a dummy procedure argument.

Now we need to define a **restricted expression**, A restricted expression is an expression in which each operation is intrinsic or defined by a specification function and each primary is

i) a constant or subobject of a constant,

ii) all or part of a dummy argument that is not optional or of intent `out`,

iii) all or part of an object that is made accessible by use or host association,[5]

iv) in a `block` construct, all or part of a local variable either of the procedure containing the construct or of an outer `block` construct containing the construct,

v) an array constructor where each *expr* is a restricted expression,

[5]Or is in a common block (Section B.3.3).

vi) a structure constructor where each component is a restricted expression,

vii) a specification inquiry where each designator or function argument is

  a. a restricted expression or

  b. a variable other than an optional argument whose properties inquired about are not dependent on the upper bound of the last dimension of an assumed-size array, deferred, or defined by an expression that is not a restricted expression,

viii) a specification inquiry where each designator or function argument is a reference to the intrinsic function `present`,

ix) a reference to any other standard intrinsic function where each argument is a restricted expression,

x) a reference to a transformational function from the intrinsic module `ieee_arithmetic`, `ieee_exceptions`, or `iso_c_binding` where each argument is a restricted expression,

xi) a reference to a specification function where each argument is a restricted expression,

xii) in a definition of a derived type, a type parameter of the type being defined,

xiii) a *variable* within an array constructor where each *expr* of the corresponding *constructor-implied-do* is a restricted expression, or

xiv) a restricted expression enclosed in parentheses,

where each subscript, section subscript, substring starting or ending point, and type parameter value is a restricted expression.

Evaluation of a specification expression in a subprogram must not cause a procedure defined by the subprogram to be invoked.

No variable referenced is allowed to have its type and type parameters specified later in the same sequence of specification statements, unless they are those implied by the implicit typing rules.

If a specification expression includes a specification inquiry that depends on a type parameter or an array bound, the type parameter or array bound must be specified earlier.[6] It may be to the left in the same statement, but not within same entity declaration. If a specification expression includes a reference to the value of an element of an array, the array shall be completely specified in prior declarations.[7]

A generic entity referenced in a specification expression in a specification sequence must have no specific procedures defined in the scoping unit, or its host scoping unit, subsequent to the specification expression.

---

[6]This avoids such a case as

```
character (len=len(a)) ::  fun
character (len=len(fun)) ::  a
```

[7]This avoids such a case as

```
integer, parameter, dimension (j(1):j(1)+1) ::  i = (/0,1/)
integer, parameter, dimension (i(1):i(1)+1) ::  j = (/1,2/)
```

An array whose bounds are declared using specification expressions is called an **explicit-shape array**.

A variety of possibilities are shown in Figure 8.12.

Figure 8.12 A variety of declarations in a subprogram.

```
subroutine sample(arr, n, string)
   use definitions  ! Contains the real a and the integer datasetsize
   integer, intent(in)               :: n
   real, dimension(n), intent(out)   :: arr    ! Explicit-shape array
   character(len=*), intent(in)      :: string ! Assumed length
   real, dimension(datasetsize+5)    :: x      ! Automatic array
   character(len=n+len(string))      :: cc     ! Automatic object
   integer, parameter :: pa2 =  selected_real_kind(2*precision(a))
   real(kind=pa2)     :: z       ! Precision of z is at least twice
                                 ! the precision of a
```

The bounds and character lengths are not affected by any redefinitions or undefinitions of variables in the expressions during execution of the procedure.

As the interfaces of specification functions must be explicit yet they cannot be internal functions,[8] such functions are probably most conveniently written as module procedures.

This feature is a great convenience for specification expressions that cannot be written as simple expressions. Here is an example:

```
function solve (a, ...
   use matrix_ops
   type(matrix), intent(in) :: a
   real                     :: work(wsize(a))
```

where `matrix` is a type defined in the module `matrix_ops` and intended to hold a sparse matrix and its LU factorization:

```
type matrix
   integer :: n            ! Matrix order.
   integer :: nz           ! Number of nonzero entries.
   logical :: new = .true. ! Whether this is a new, unfactorized
                           ! matrix.
      ⋮
end type matrix
```

and `wsize` is a module procedure that calculates the required size of the array `work`:

```
pure integer function wsize(a)
   type(matrix), intent(in) :: a
   wsize = 2*a%n + 2
   if(a%new) wsize = a%nz + wsize
end function wsize
```

[8] This prevents them inquiring, via host association, about objects being specified in the set of statements in which the specification function itself is referenced.

Note that a dummy argument of an elemental subprogram is permitted to be used in a specification expression for a local variable. Here is a partial example:

```
elemental real function f(a, b, order)
  real, intent (in)    :: a, b
  integer, intent (in) :: order
  real                 :: temp(order)
  ⋮
```

In this elemental function, the local variable `temp` is an array whose size depends on the `order` argument.

8.19 Structure constructors

Like procedures, structure constructors can have keyword arguments and optional arguments; moreover, a generic procedure name can be the same as the structure constructor name (which is the same as the type name), with any specific procedures in the generic set taking precedence over the structure constructor if there is any ambiguity. This can be used effectively to produce extra 'constructors' for the type, as shown in Figure 8.13.

Keyword arguments permit a parent (or ancestor) component to be specified in a structure constructor, as long as it does not result in a component being given a value more than once.

If a component of a type has default initialization or is an allocatable component, its value may be omitted in the structure constructor as if it were an optional argument. For example, in

```
type real_list_element
   real                              :: value
   type(real_list_element), pointer :: next => null()
end type real_list_element
⋮
type(real_list_element) :: x = real_list_element(3.5)
```

the omitted value for the `next` component means that it takes on its default initialization value – that is, a null pointer. For an allocatable component, it is equivalent to specifying `null()` for that component value.

8.20 The namelist statement

It is sometimes convenient to gather a set of variables into a single group in order to facilitate input/output (I/O) operations on the group as a whole. The actual use of such groups is explained in Section 10.10. The method by which a group is declared is via the `namelist` statement, which in its simple form has the syntax

`namelist` *namelist-spec*

where *namelist-spec* is

/*namelist-group-name*/ *variable-name-list*

Figure 8.13 Keywords in a structure constructor and a function as a structure constructor.

```
module mycomplex_module
   type mycomplex
      real :: argument, modulus
   end type
   interface mycomplex
      module procedure complex_to_mycomplex, two_reals_to_mycomplex
   end interface
   ⋮
contains
   type(mycomplex) function complex_to_mycomplex(c)
      complex, intent(in) :: c
      ⋮
   end function complex_to_mycomplex
   type(mycomplex) function two_reals_to_mycomplex(x, y)
      real, intent(in)           :: x
      real, intent(in), optional :: y
      ⋮
   end function two_reals_to_mycomplex
   ⋮
end module mycomplex_module
⋮
use mycomplex_module
type(mycomplex) :: a, b, c
⋮
a = mycomplex(argument=5.6, modulus=1.0) ! The structure constructor
c = mycomplex(x=0.0, y=1.0)              ! A function reference
```

The *namelist-group-name* is the name given to the group for subsequent use in the I/O statements. An example is

```
real :: carpet, tv, brushes(10)
namelist /household_items/ carpet, tv, brushes
```

It is possible to declare several namelist groups in one statement, with the syntax

`namelist` *namelist-spec [[,] namelist-spec]* ...

as in the example

```
namelist /list1/ a, b, c /list2/ x, y, z
```

It is possible to continue a list within the same scoping unit by repeating the namelist name on more than one statement. Thus,

```
namelist /list/ a, b, c
namelist /list/ d, e, f
```

has the same effect as a single statement containing all the variable names in the same order. A namelist group object may appear more than once in a namelist group and may belong to more than one namelist group.

If the type, kind type parameters, or rank of a namelist variable is specified in a specification statement in the same scoping unit, the specification statement must either appear before the `namelist` statement, or be a type declaration statement that confirms the implicit typing rule in force in the scoping unit for the initial letter of the variable. Also, if the namelist group has the `public` attribute, no variable in the list may have the `private` attribute or have private components.[9]

8.21 Summary

In this chapter most of the specification statements of Fortran have been described. The following concepts have been introduced: implicit typing and its attendant dangers, named constants, constant expressions, data initialization, control of the accessibility of entities in modules, saving data between procedure calls, volatility, selective access of entities in a module, renaming entities accessed from a module, specification expressions that may be used when specifying data objects and function results, and the formation of variables into namelist groups. We have also explained alternative ways of specifying attributes.

We conclude this chapter with a complete program, Figure 8.14, that uses a module to sort US-style addresses (name, street, town, and state with a numerical zip code) in order of zip code. It illustrates the interplay between many of the features described so far, but note that it is not production code since the sort subroutine is not very efficient and the full range of US addresses is not handled. Suitable test data are:

```
Prof. James Bush,
206 Church St. SE,
Minneapolis,
MN 55455

J. E. Dougal,
Rice University,
Houston,
TX 77251

Jack Finch,
104 Ayres Hall,
Knoxville,
TN 37996
```

[9] A variable that is an assumed-size array (Section 19.5) is prohibited.

Figure 8.14 A module to sort postal addresses and a program that uses it; `maxloc` is described in Section 9.16. The `read` and `write` statements here are explained in Section 10.7 and Chapter 11.

```
module sort           ! To sort postal addresses by zip code.
   implicit none
   private
   public :: selection_sort
   integer, parameter :: string_length = 30
   type, public :: address
      character(len = string_length) :: name, street, town, &
                                        state*2
      integer                        :: zip_code
   end type address
contains
   recursive subroutine selection_sort (array_arg)
      type (address), dimension (:), intent (inout)         &
                                           :: array_arg
      integer                              :: current_size
      integer                              :: big
      current_size = size (array_arg)
      if (current_size > 0) then
         big = maxloc (array_arg(:)%zip_code, dim=1)
         call swap (big, current_size)
         call selection_sort (array_arg(1: current_size - 1))
      end if
   contains
      subroutine swap (i, j)
         integer, intent (in) :: i, j
         type (address)       :: temp
         temp = array_arg(i)
         array_arg(i) = array_arg(j)
         array_arg(j) = temp
      end subroutine swap
   end subroutine selection_sort
end module sort
program zippy
   use sort
   implicit none
   integer, parameter                     :: array_size = 100
   type (address), dimension (array_size) :: data_array
   integer                                :: i, n
   do i = 1, array_size
      read  (*, '(/a/a/a/a2,i8)', end=10) data_array(i)
      write (*, '(/a/a/a/a2,i8)')         data_array(i)
   end do
10 n = i - 1
   call selection_sort (data_array(1: n))
   write (*, '(//a)') 'after sorting:'
   do i = 1, n
      write (*, '(/a/a/a/a2,i8)') data_array(i)
   end do
end program zippy
```

Exercises

1. Write suitable type statements for the following quantities:

i) an array to hold the number of counts in each of the 100 bins of a histogram numbered from 1 to 100;

ii) an array to hold the temperature to two decimal places at points, on a sheet of iron, equally spaced at 1 cm intervals on a rectangular grid 20 cm square, with points in each corner (the melting point of iron is 1530 °C);

iii) an array to describe the state of 20 on/off switches;

iv) an array to contain the information destined for a printed page of 44 lines, each of 70 letters or digits.

2. Explain the difference between the following pair of declarations:

```
real :: i = 3.1
```

and

```
real, parameter :: i = 3.1
```

What is the value of `i` in each case?

3. Write type declaration statements which initialize:

i) all the elements of an integer array of length 100 to the value zero;

ii) all the odd elements of the same array to 0 and the even elements to 1;

iii) the elements of a real 10×10 square array to 1.0;

iv) a character string to the digits '0' to '9'.

4. i) Write a type declaration statement that declares and initializes a variable of derived type `person` (Section 2.9).

ii) Either

a. write a type declaration statement that declares and initializes a variable of type `entry` (Section 2.12); or

b. write a type declaration statement for such a variable and a `data` statement to initialize its non-pointer components.

5. Which of the following are constant expressions?

i) `kind(x)`, for `x` of type real

ii) `selected_real_kind(6, 20)`

iii) `1.7**2`

iv) `1.7**2.0`

v) `(1.7, 2.3)**(-2)`

vi) `(/ (7*i, i=1, 10) /)`

vii) `person("Reid", 25*2.0, 22**2)`

viii) `entry(1.7, 1, null_pointer)`

9. Intrinsic procedures and modules

9.1 Introduction

In a language that has a clear orientation towards scientific applications, there is an obvious requirement for the most frequently required mathematical functions to be provided as part of the language itself, rather than expecting each user to code them afresh. When provided with the compiler, they are normally coded to be very efficient and will have been well tested over the complete range of values that they accept. It is difficult to compete with the high standard of code provided by the vendors.

The efficiency of the intrinsic procedures when handling arrays is particularly marked because a single call may cause a large number of individual operations to be performed, during the execution of which advantage may be taken of the specific nature of the hardware.

Another feature of a substantial number of the intrinsic procedures is that they extend the power of the language by providing access to facilities that are not otherwise available. Examples are inquiry functions for the presence of an optional argument, the parts of a floating-point number, and the length of a character string.

There are over 170 intrinsic procedures in all, a particularly rich set. They fall into distinct groups, each of which we describe in turn. Some processors may offer additional intrinsic procedures. Note that a program containing references to such procedures is portable only to other processors that provide those same procedures. In fact, such a program does not conform to the standard.

All the intrinsic procedures are generic.

9.1.1 Keyword calls

The procedures may be called with keyword actual arguments, using the dummy argument names as keywords. This facility is not very useful for those with a single non-optional argument, but is useful for those with several optional arguments. For example,

```
call date_and_time (date=d)
```

returns the date in the scalar character variable `d`. The rules for positional and keyword argument lists were explained in Section 5.13. In this chapter, the dummy arguments that are optional are indicated with square brackets. We have taken some ‘poetic licence’ with this notation, which might suggest to the reader that the positional form is permitted following an absent argument (this is not the case).

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

9.1.2 Categories of intrinsic procedures

There are four categories of intrinsic procedures.

i) **Elemental procedures** (Section 7.4).

ii) **Inquiry functions** return properties of their principal arguments that do not depend on their values; indeed, for variables, their values may be undefined.

iii) **Transformational functions** are functions that are neither elemental nor inquiry; they usually have array arguments and an array result whose elements depend on many of the elements of the arguments.

iv) **Non-elemental subroutines**.

All the functions are pure (Section 7.8). The subroutine `mvbits` is pure. The subroutine `move_alloc` is pure unless it is applied to coarrays.

9.1.3 The intrinsic statement

A name may be specified to be that of an intrinsic procedure in an `intrinsic` statement, which has the general form

`intrinsic` *`[::]`* *intrinsic-name-list*

where *intrinsic-name-list* is a list of intrinsic procedure names. A name must not appear more than once in the `intrinsic` statements of a scoping unit and must not appear in an `external` statement there (but may appear as a generic name on an interface block if an intrinsic procedure is being extended, see Section 5.18). It is possible to include such a statement in every scoping unit that contains references to intrinsic procedures, in order to make the use clear to the reader. We particularly recommend this practice when referencing intrinsic procedures that are not defined by the standard, for then a clear diagnostic message should be produced if the program is ported to a processor that does not support the extra intrinsic procedures.

9.1.4 Argument intents

Since all the functions are pure, their arguments all have intent `in`. For the subroutines, the intents vary from case to case (see the descriptions given later in the chapter).

9.2 Inquiry functions for any type

The following are inquiry functions whose arguments may be of any type.

`allocated (array)` or **`allocated (scalar)`** returns, when the allocatable array `array` or the allocatable scalar `scalar` is currently allocated, the value true; otherwise it returns the value false.

associated (pointer *[*,target*]*), when target is absent, returns the value true if the pointer pointer is associated with a target and false otherwise. The pointer association status of pointer must not be undefined. If target is present, it must be allowable as a target of pointer. The value is true if pointer is associated with target, and false otherwise.

In the array case, true is returned only if the shapes are identical and corresponding array elements, in array element order, are associated with each other. If the character length or array size is zero, false is returned. A different bound, as in the case of associated(p,a) following the pointer assignment p => a(:) when lbound(a) = 0, is insufficient to cause false to be returned.

The argument target may itself be a pointer, in which case its target is compared with the target of pointer; the pointer association status of target must not be undefined and if either pointer or target is disassociated, the result is false.

If target is an internal procedure or a pointer associated with an internal procedure, the result is true only if pointer and target also have the same host instance.

present (a) may be called in a subprogram that has an optional dummy argument a or accesses such a dummy argument from its host. It returns the value true if the corresponding actual argument is present in the current call to it, and false otherwise. If an absent dummy argument is used as an actual argument in a call of another subprogram, it is regarded as also absent in the called subprogram.

There is an inquiry function whose argument may be of any intrinsic type:

kind (x) has type default integer and value equal to the kind type parameter value of x.

9.3 Elemental numeric functions

There are 17 elemental functions for performing simple numerical tasks, many of which perform type conversions for some or all permitted types of arguments.

9.3.1 Elemental functions that may convert

If kind is present in the following elemental functions, it must be a scalar integer constant expression and provide a kind type parameter that is supported on the processor.

abs (a) returns the absolute value of an argument of type integer, real, or complex. The result is of type integer if a is of type integer and otherwise it is real. It has the same kind type parameter as a.

aimag (z) returns the imaginary part of the complex value z. The type of the result is real and its kind type parameter is that of z.

aint (a *[*,kind*]*) truncates a real value a towards zero to produce a real that is a whole number. The value of the kind type parameter of the result is the value of the argument kind if it is present, or that of a otherwise.

anint (a *[,kind]*) returns a real whose value is the nearest whole number to the real value `a`. The value of the kind type parameter of the result is the value of the argument `kind`, if it is present, or that of `a` otherwise.

ceiling (a *[,kind]*) returns the least integer greater than or equal to its real argument. If `kind` is present, the value of the kind type parameter of the result is the value of `kind`, otherwise it is that of the default integer type.

cmplx (x *[,y, kind]*) converts `x` or `(x, y)` to complex type with the value of the kind type parameter of the result being the value of `kind` if it is present or of default complex otherwise. The argument `x` may be of type integer, real, or complex, or it may be a 'boz' constant. If `x` is of type complex, `y` must be absent and `kind` must be given by keyword if it is wanted.[1] If `y` is absent and `x` is not of type complex, the result is as if `y` were present with the value zero. If `x` is of type complex, the result is as if `x` were present with the value `real(x,kind)` and `y` were present with the value `aimag(x,kind)`. The value of `cmplx(x,y,kind)` has real part `real(x,kind)` and imaginary part `real(y,kind)`.

floor (a *[,kind]*) returns the greatest integer less than or equal to its real argument. If `kind` is present, the value of the kind type parameter of the result is the value of `kind`, otherwise it is that of the default integer type.

int (a *[,kind]*) converts to integer type with the value of the kind type parameter being the value of the argument `kind`, if it is present, or that of the default integer otherwise. The argument `a` may be

- integer, in which case `int(a) = a`;
- real, in which case the value is truncated towards zero;
- complex, in which case the real part is truncated towards zero; or
- a 'boz' constant, in which case the result has the bit sequence of the specified integer after truncation on the left or padding with zeros on the left to the bit length of the result; if the leading bit is 1, the value is processor dependent.

nint (a *[,kind]*) returns the integer value that is nearest to the real `a`, or the even such value if two are equally near. If `kind` is present, the value of the kind type parameter of the result is the value of `kind`, otherwise it is that of the default integer type.

real (a *[,kind]*) converts to real type with the value of the kind type parameter being that of `kind` if it is present. If `kind` is absent, the kind type parameter is that of `a` if `a` is of type complex and default real otherwise. The argument `a` may be

- integer or real, in which case the result is a processor-dependent approximation to `a`;

[1] This requirement is removed in Fortran 2018, see Section 23.7.3.

- complex, in which case the result is a processor-dependent approximation to the real part of `a`; or
- a 'boz' constant, in which case the result has the bit sequence of the specified integer after truncation on the left or padding with zeros on the left to the bit length of the result.

9.3.2 Elemental functions that do not convert

The following are elemental functions whose result is of type and kind type parameter that are those of the first or only argument. For those having more than one argument, all arguments must have the same type and kind type parameter.

`conjg (z)` returns the conjugate of the complex value `z`.

`dim (x, y)` returns `max(x-y, 0.)` for arguments that are both integer or both real.

`max (a1, a2 [,a3,...] )` returns the maximum of two or more values, which must all be integer, all be real, or all be character (character comparison is explained in Section 3.5).

`min (a1, a2 [,a3,...] )` returns the minimum of two or more values, which must all be integer, all be real, or all be character (character comparison is explained in Section 3.5).

`mod (a, p)` returns the remainder of `a` modulo `p`, that is `a-int(a/p)*p`. The value of `p` must not be zero; `a` and `p` must be both integer or both real.

`modulo (a, p)` returns `a` modulo `p` when `a` and `p` are both integer or both real, that is `a-floor(a/p)*p` in the real case, and `a-floor(a÷p)*p` in the integer case, where ÷ represents ordinary mathematical division. The value of `p` must not be zero.

`sign (a, b)` returns the absolute value of `a` times the sign of `b`. The arguments `a` and `b` must be both integer or both real.[2] If `b` is real with the value zero and the processor can distinguish between a negative and a positive real zero, the result has the sign of `b` (see also Section 9.9.1). Otherwise, if `b` is zero, its sign is taken as positive.

9.4 Elemental mathematical functions

The following are elemental functions that evaluate elementary mathematical functions. The type and kind type parameter of the result are those of the first argument, which is usually the only argument.

`acos (x)` returns the arc cosine (inverse cosine) function value for real or complex values x such that $|x| \leq 1$, expressed in radians in the range $0 \leq \text{acos}(x) \leq \pi$.

[2] This restriction is relaxed in Fortran 2018, see Section 23.7.1.

acosh (x) returns the inverse hyperbolic cosine for real or complex values of x, that is, `y` such that `cosh(y)` would be approximately equal to `x`.

asin (x) returns the arc sine (inverse sine) function value for real or complex values x such that $|x| \leq 1$, expressed in radians such that $-\frac{\pi}{2} \leq$ `real(asin(x))` $\leq \frac{\pi}{2}$.

asinh (x) returns the inverse hyperbolic sine for real or complex values of x.

atan (x) returns the arc tangent (inverse tangent) function value for real or complex x, whose real part is expressed in radians in the range $-\frac{\pi}{2} \leq$ `real(atan(x))` $\leq \frac{\pi}{2}$.

atan (y, x) is the same as `atan2 (y, x)`, see below.

atan2 (y, x) returns the arc tangent (inverse tangent) function value for pairs of reals, x and y, of the same type and type parameter. The result is the principal value of the argument of the complex number (x,y), expressed in radians in the range $-\pi \leq \text{atan2}(y,x) \leq \pi$. If the processor can distinguish between positive and negative real zero (e.g. if the arithmetic is IEEE), an approximation to $-\pi$ is returned if $x < 0$ and y is a negative zero. If x is zero, the absolute value of the result is approximately $\frac{\pi}{2}$.

atanh (x) returns the inverse hyperbolic tangent for real or complex values of x. If the result is complex, the imaginary part is expressed in radians and it satisfies the inequality $-\frac{\pi}{2} \leq$ `aimag(atanh(x))` $\leq \frac{\pi}{2}$.

bessel_j0 (x) returns the Bessel function of first kind and order zero; `x` must be of type real, and `n` must be of type integer with a non-negative value.

bessel_j1 (x) returns the Bessel function of first kind and order one for a real argument.

bessel_jn (n, x) returns the Bessel function of first kind and order `n`; `x` must be of type real, and `n` must be of type integer with a non-negative value. See Section 9.5 for **bessel_jn (n1, n2, x)**.

bessel_y0 (x) returns the Bessel function of second kind and order zero for a real argument.

bessel_y1 (x) returns the Bessel function of second kind and order one for a real argument whose value is greater than zero.

bessel_yn (n, x) returns the Bessel function of second kind and order `n`; `x` must be of type real, and `n` must be of type integer with a non-negative value. See Section 9.5 for **bessel_yn (n1, n2, x)**.

cos (x) returns the cosine function value for an argument of type real or complex. The result, or its real part if complex, is treated as a value in radians.

cosh (x) returns the hyperbolic cosine function value for a real or complex argument x. If `x` is complex, the imaginary part is treated as a value in radians.

erf (x) returns the value of the error function of `x`, $\frac{2}{\sqrt{\pi}}\int_0^{\mathtt{x}} e^{-t^2}\,dt$ for `x` of type real.

erfc (x) returns the complement of the error function, $1 - \mathtt{erf(x)}$, for `x` of type real. This has the mathematical form $\frac{2}{\sqrt{\pi}}\int_{\mathtt{x}}^{\infty} e^{-t^2}\,dt$.

erfc_scaled (x) returns the exponentially scaled error function, `exp(x**2)*erfc(x)`, for `x` of type real. Note that as `x` increases its error function very rapidly approaches one, and its complement quite quickly approaches zero. In IEEE double precision, `erf(x)` is equal to one for $\mathtt{x} > 6$ and `erfc(x)` underflows for $\mathtt{x} > 26.7$. Thus `erfc_scaled` may be more useful than `erfc` when `x` is not small.

exp (x) returns the exponential function value for a real or complex argument `x`. If `x` is complex, the imaginary part of the result is expressed in radians.

gamma (x) returns the value of the gamma function at `x`; `x` must be of type real with a value that is not zero or a negative whole number.

hypot (x, y) returns the Euclidean distance, that is $\sqrt{\mathtt{x}^2 + \mathtt{y}^2}$, calculated without undue overflow or underflow. The arguments `x` and `y` must be of type real with the same kind type parameter, and the result is also real of that kind.[3]

log (x) returns the natural logarithm function for a real or complex argument x. In the real case, x must be positive. In the complex case, x must not be zero, and the imaginary part w of the result lies in the range $-\pi \leq w \leq \pi$. If the processor can distinguish between positive and negative real zero (e.g. if the arithmetic is IEEE), an approximation to $-\pi$ is returned if the real part of x is less than zero and the imaginary part is a negative zero.

log_gamma (x) returns the natural logarithm of the absolute value of the gamma function, `log(abs(gamma(x)))`; `x` must be of type real with a value that is not zero or a negative whole number.

log10 (x) returns the common (base 10) logarithm of a real argument whose value must be positive.

sin (x) returns the sine function value for a real or complex argument. If `x` is complex, the real part of the result is treated as a value in radians.

sinh (x) returns the hyperbolic sine function value for a real or complex argument. If `x` is complex, the imaginary part of the result is treated as a value in radians.

sqrt (x) returns the square root function value for a real or complex argument `x`. If `x` is real, its value must not be negative. In the complex case, the real part of the result is not negative, and when it is zero the imaginary part is not negative. If the arithmetic is IEEE, a negative imaginary result is returned if the real part of the result is zero and the imaginary part of `x` is less than zero.

[3] This intrinsic function means that there are three intrinsic functions that calculate Euclidean distances, which seems a trifle unnecessary for such a simple thing. (The other two functions are `abs(cmplx(x, y))`, which has been available for this purpose since Fortran 90, and `norm2([x, y])`.)

tan (x) returns the tangent function value for a real or complex argument. If `x` is real, the result is treated as a value in radians. If `x` is complex, the real part of the result is treated as a value in radians.

tanh (x) returns the hyperbolic tangent function value for a real or complex argument. If `x` is complex, the imaginary part of the result is treated as a value in radians.

9.5 Transformational functions for Bessel functions

Two transformational functions return rank-one arrays of multiple Bessel function values:

bessel_jn (n1, n2, x) first kind and orders `n1` to `n2`;

bessel_yn (n1, n2, x) second kind and orders `n1` to `n2`.

In this case `n1` and `n2` must be of type integer with non-negative values, and all three arguments must be scalar. If `n2` $<$ `n1`, the result has zero size.

It is potentially more efficient to calculate successive Bessel function values together rather than separately, so if these are required the transformational forms should be used instead of multiple calls to the elemental ones.

9.6 Elemental character and logical functions

9.6.1 Character–integer conversions

The following are elemental functions for conversions from a single character to an integer, and vice versa.

achar (i *[,kind]*) is of type character with length one and returns the character in the position in the ASCII collating sequence that is specified by the integer `i`. The value of `i` must be in the range $0 \leq i \leq 127$, otherwise the result is processor dependent. The optional `kind` argument specifies the kind of the result, which is of default kind if it is absent. For instance, if the processor had an extra character kind 37 for EBCDIC, `achar(iachar('h'),37)` would return the EBCDIC lower-case `'h'` character.

char (i *[,kind]*) is of type character and length one, with kind type parameter value that of the value of `kind` if present, or default otherwise. It returns the character in position `i` in the processor collating sequence associated with the relevant kind parameter. The value of `i` must be in the range $0 \leq i \leq n-1$, where n is the number of characters in the processor's collating sequence. If `kind` is present, it must be a scalar integer constant expression and provide a kind type parameter that is supported on the processor.

iachar (c *[,kind]*) is of type integer and returns the position in the ASCII collating sequence of the character `c` if it is in the ASCII character set and a processor-dependent value otherwise. The optional final `kind` argument must be a scalar integer constant expression and, if present, it specifies the kind of the result, which is otherwise of default kind.

ichar (c *[,kind]*) is of type integer and returns the position of the character `c` in the processor collating sequence associated with the kind parameter of `c`. The optional final `kind` argument must be a scalar integer constant expression. If present, it specifies the kind of the result, which is otherwise of default kind.

9.6.2 Lexical comparison functions

The following elemental functions accept arguments that are of type character and are both of default kind or both of ASCII kind, make a lexical comparison based on the ASCII collating sequence, and return a default logical result. If the strings have different lengths, the shorter one is padded on the right with blanks.

lge (string_a, string_b) returns the value true if `string_a` follows `string_b` in the ASCII collating sequence or is equal to it, and the value false otherwise.

lgt (string_a, string_b) returns the value true if `string_a` follows `string_b` in the ASCII collating sequence, and the value false otherwise.

lle (string_a, string_b) returns the value true if `string_b` follows `string_a` in the ASCII collating sequence or is equal to it, and the value false otherwise.

llt (string_a, string_b) returns the value true if `string_b` follows `string_a` in the ASCII collating sequence, and false otherwise.

9.6.3 String-handling elemental functions

The following are elemental functions that manipulate strings. The arguments `string`, `substring`, and `set` are always of type character, and where two are present have the same kind type parameter. The kind type parameter value of the result is that of `string`.

adjustl (string) adjusts left to return a string of the same length by removing all leading blanks and inserting the same number of trailing blanks.

adjustr (string) adjusts right to return a string of the same length by removing all trailing blanks and inserting the same number of leading blanks.

index (string, substring *[,back] [,kind]*) returns an integer holding the starting position of `substring` as a substring of `string`, or zero if it does not occur as a substring. If `back` is absent or present with value false, the starting position of the first such substring is returned; the value 1 is returned if `substring` has zero length. If `back` is present with value true, the starting position of the last such substring is returned; the value `len(string)+1` is returned if `substring` has zero length. If `kind` is present, it specifies the kind of the result, which is otherwise default; it must be a scalar integer constant expression.

len_trim (string *[,kind]*) returns an integer whose value is the length of `string` without trailing blank characters. If `kind` is present, it specifies the kind of the result, which is otherwise default; it must be a scalar integer constant expression.

scan (string, set *[*,back*]* *[*,kind*]*) returns an integer whose value is the position of a character of `string` that is in `set`, or zero if there is no such character. If the logical `back` is absent or present with value false, the position of the leftmost such character is returned. If `back` is present with value true, the position of the rightmost such character is returned. If `kind` is present, it specifies the kind of the result, which is otherwise default; it must be a scalar integer constant expression.

verify (string, set *[*,back*]* *[*,kind*]*) returns the integer value 0 if each character in `string` appears in `set`, or the position of a character of `string` that is not in `set`. If the logical `back` is absent or present with value false, the position of the leftmost such character is returned. If `back` is present with value true, the position of the rightmost such character is returned. If `kind` is present, it specifies the kind of the result, which is otherwise default; it must be a scalar integer constant expression.

9.6.4 Logical conversion

The following elemental function converts from a logical value with one kind type parameter to another.

logical (l *[*,kind*]*) returns a logical value equal to the value of the logical `l`. The value of the kind type parameter of the result is the value of `kind` if it is present or that of default logical otherwise. If `kind` is present, it must be a scalar integer constant expression and provide a kind type parameter that is supported on the processor.

9.7 Non-elemental string-handling functions

9.7.1 String-handling inquiry function

len (string *[*,kind*]*) is an inquiry function that returns a scalar integer holding the number of characters in `string` if it is scalar, or in an element of `string` if it is array valued. The value of `string` need not be defined. If `kind` is present, it specifies the kind of the result, which is otherwise default; it must be a scalar integer constant expression and provide a kind type parameter that is supported on the processor.

9.7.2 String-handling transformational functions

There are two functions that cannot be elemental because the length type parameter of the result depends on the value of an argument.

repeat (string, ncopies) forms the string consisting of the concatenation of `ncopies` copies of `string`, where `ncopies` is of type integer and its value must not be negative. Both arguments must be scalar.

trim (string) returns `string` with all trailing blanks removed. The argument `string` must be scalar.

9.8 Character inquiry function

The intrinsic inquiry function `new_line(a)` returns the character that can be used to cause record termination (this is the equivalent of the C language `'\n'` character):

`new_line (a)` returns the newline character used for formatted stream output. The argument `a` must be of type character. The result is of type `character` with the same kind type parameter value as `a`. In the unlikely event that there is no suitable character for newline in that character set, a blank is returned.

9.9 Numeric inquiry and manipulation functions

9.9.1 Models for integer and real data

The numeric inquiry and manipulation functions are defined in terms of a model set of integers and a model set of reals for each kind of integer and real data type implemented. For each kind of integer, it is the set

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

where s is ± 1, q is a positive integer, r is an integer exceeding 1 (usually 2), and each w_k is an integer in the range $0 \leq w_k < r$. For each kind of real, it is the set

$$x = 0$$

and

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}$$

where s is ± 1, p and b are integers exceeding 1, e is an integer in a range $e_{\min} \leq e \leq e_{\max}$, and each f_k is an integer in the range $0 \leq f_k < b$ except that f_1 is also nonzero.

Values of the parameters in these models are chosen for the processor so as best to fit the hardware with the proviso that all model numbers are representable. Note that it is quite likely that there are some machine numbers that lie outside the model. For example, many computers represent the integer $-r^q$, and the IEEE standard for floating-point arithmetic (ISO/IEC/IEEE 60559:2011) contains reals with $f_1 = 0$ (called subnormal numbers) and register numbers with increased precision and range.

In the first paragraph of Section 2.6, we noted that the value of a signed zero is regarded as being the same as that of an unsigned zero. However, many processors distinguish at the hardware level between a negative real zero value and a positive real zero value, and the IEEE standard makes use of this where possible. For example, when the exact result of an operation is nonzero but the rounding produces a zero, the sign is retained.

In Fortran, the two zeros are treated identically in all relational operations, as input arguments to all intrinsic functions (except `sign`), or as the scalar expression in the arithmetic `if` statement (Appendix B.6). However, the function `sign` (Section 9.3.2) is such that the sign of the second argument may be taken into account even if its value is zero. On a processor

that has IEEE arithmetic, the value of `sign(2.0, -0.0)` is -2.0. Also, a Fortran processor is required to represent all negative numbers on output, including zero, with a minus sign.

9.9.2 Numeric inquiry functions

There are nine inquiry functions that return values from the models associated with their arguments. Each has a single argument that may be scalar or array valued and each returns a scalar result. The value of the argument need not be defined.

`digits (x)`, for real or integer `x`, returns the default integer whose value is the number of significant digits in the model that includes `x`, that is p or q.

`epsilon (x)`, for real `x`, returns a real result with the same type parameter as `x` that is almost negligible compared with the value 1.0 in the model that includes `x`, that is b^{1-p}.

`huge (x)`, for real or integer `x`, returns the largest value in the model that includes `x`. It has the type and type parameter of `x`. The value is

$$(1-b^{-p})b^{e_{\max}}$$

or

$$r^q - 1$$

`maxexponent (x)`, for real `x`, returns the default integer $e_{\max}$, the maximum exponent in the model that includes `x`.

`minexponent (x)`, for real `x`, returns the default integer $e_{\min}$, the minimum exponent in the model that includes `x`.

`precision (x)`, for real or complex `x`, returns a default integer holding the equivalent decimal precision in the model representing real numbers with the same type parameter value as `x`. The value is

$$\texttt{int}((p-1)*\texttt{log10}(b))+k$$

where k is 1 if b is an integral power of 10 and 0 otherwise.

`radix (x)`, for real or integer `x`, returns the default integer that is the base in the model that includes `x`, that is b or r.

`range (x)`, for integer, real, or complex `x`, returns a default integer holding the equivalent decimal exponent range in the models representing integer or real numbers with the same type parameter value as `x`. The value is `int(log10(`*huge*`))` for integers and

```
int(min(log10(huge), -log10(tiny)))
```

for reals, where *huge* and *tiny* are the largest and smallest positive numbers in the models.

tiny (x), for real x, returns the smallest positive number

$$b^{e_{\min}-1}$$

in the model that includes x. It has the type and type parameter of x.

9.9.3 Elemental functions to manipulate reals

There are seven elemental functions whose first or only argument is of type real and that return values related to the components of the model values associated with the actual value of the argument. For the functions exponent, fraction, and set_exponent, if the value of x lies outside the range of model numbers, its e value is determined as if the model had no exponent limits.

exponent (x) returns the default integer whose value is the exponent part e of x when represented as a model number. If x = 0, the result has value zero.

fraction (x) returns a real with the same type parameter as x whose value is the fractional part of x when represented as a model number, that is $\mathrm{x}\, b^{-e}$.

nearest (x, s) returns a real with the same type parameter as x whose value is the nearest different machine number in the direction given by the sign of the real s. The value of s must not be zero.

rrspacing (x) returns a real with the same type parameter as x whose value is the reciprocal of the relative spacing of model numbers near x. If the value of x is a model number this is $|\mathrm{x}\, b^{-e}|b^{p}$.

scale (x, i) returns a real with the same type parameter as x, whose value is $\mathrm{x}\, b^{\mathrm{i}}$, where b is the base in the model for x, and i is of type integer.

set_exponent (x, i) returns a real with the same type parameter as x, whose fractional part is the fractional part of the model representation of x and whose exponent part is i, that is $\mathrm{x}\, b^{\mathrm{i}-e}$.

spacing (x) returns a real with the same type parameter as x whose value is the absolute spacing of model numbers near x.

9.9.4 Transformational functions for kind values

There are three functions that return the least kind type parameter value that will meet a given requirement. They have scalar arguments and results, so are classified as transformational.

selected_char_kind (name) returns the kind value for the character set whose name is given by the character string name, or -1 if it is not supported (or if the name is not recognized). In particular, if name is

default, the result is the kind of the default character type (equal to kind('A'));

`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.

Other character set names are processor dependent. The character set name is not case sensitive (lower case is treated as upper case), and any trailing blanks are ignored.

Note that the only character set which is guaranteed to be supported is the default character set; a processor is not required to support ASCII or ISO 10646.

`selected_int_kind (r)` returns the default integer scalar that is the kind type parameter value for an integer data type able to represent all integer values n in the range $-10^r < n < 10^r$, where `r` is a scalar integer. If more than one is available, a kind with least decimal exponent range is chosen (and least kind value if several have least decimal exponent range). If no corresponding kind is available, the result is -1.

`selected_real_kind (`*`[p] [, r] [, radix]`*`)` returns the default integer scalar that is the kind type parameter value for a real data type with decimal precision (as returned by the function `precision`) at least `p`, decimal exponent range (as returned by the function `range`) at least `r`, and radix (as returned by the function `radix`) `radix`. If more than one is available, a kind with the least decimal precision is chosen (and least kind value if several have least decimal precision). All three arguments are scalar integers; at least one of them must be present. If `radix` is absent, there is no requirement on the radix selected. If no corresponding kind value is available, the result is -1 if sufficient precision is unavailable, -2 if sufficient exponent range is unavailable, -3 if both are unavailable, -4 if the radix is available with the precision or the range but not both, and -5 if the radix is not available at all.

9.10 Bit manipulation procedures

9.10.1 Model for bit data

There are intrinsic procedures for manipulating bits held within integers. They are based on a model in which an integer holds s bits w_k, $k = 0, 1, \ldots, s-1$, in a sequence from right to left, based on the non-negative value

$$\sum_{k=0}^{s-1} w_k \times 2^k$$

This model is valid only in the context of these intrinsics. It is identical to the model for integers in Section 9.9.1 when $r = 2$ and $w_{s-1} = 0$, but when $r \neq 2$ or $w_{s-1} = 1$ the models do not correspond, and the value expressed as an integer may vary from processor to processor.

9.10.2 Inquiry function

`bit_size (i)` returns the number of bits in the model for bits within an integer of the same type parameter as `i`. The result is a scalar integer having the same type parameter as `i`.

9.10.3 Basic elemental functions

btest (i, pos) returns the default logical value true if bit `pos` of the integer `i` has value 1 and false otherwise; `pos` must be an integer with value in the range $0 \leq$ `pos` $<$ `bit_size(i)`.

iand (i, j) returns the logical and of all the bits in `i` and corresponding bits in `j`, according to the truth table

`i`	1	1	0	0
`j`	1	0	1	0
`iand(i, j)`	1	0	0	0

The arguments `i` and `j` must have the same type parameter value, which is the type parameter value of the result.

ibclr (i, pos) returns an integer, with the same type parameter as `i`, and value equal to that of `i` except that bit `pos` is cleared to 0. The argument `pos` must be an integer with value in the range $0 \leq$ `pos` $<$ `bit_size(i)`.

ibits (i, pos, len) returns an integer, with the same type parameter as `i`, and value equal to the `len` bits of `i` starting at bit `pos` right adjusted and all other bits zero. The arguments `pos` and `len` must be integers with non-negative values such that `pos+len` $\leq$ `bit_size(i)`.

ibset (i, pos) returns an integer, with the same type parameter as `i`, and value equal to that of `i` except that bit `pos` is set to 1. The argument `pos` must be an integer with value in the range $0 \leq$ `pos` $<$ `bit_size(i)`.

ieor (i, j) returns the logical exclusive or of all the bits in `i` and corresponding bits in `j`, according to the truth table

`i`	1	1	0	0
`j`	1	0	1	0
`ieor(i, j)`	0	1	1	0

The arguments `i` and `j` must have the same type parameter value, which is the type parameter value of the result.

ior (i, j) returns the logical inclusive or of all the bits in `i` and corresponding bits in `j`, according to the truth table

`i`	1	1	0	0
`j`	1	0	1	0
`ior(i, j)`	1	1	1	0

The arguments `i` and `j` must have the same type parameter value, which is the type parameter value of the result.

not (i) returns the logical complement of all the bits in `i`, according to the truth table

`i`	1	0
`not(i)`	0	1

9.10.4 Shift operations

There are seven elemental functions for bit shifting: five described here, and two 'double-width' functions described in Section 9.10.7.

ishft (i, shift) returns an integer, with the same type parameter as `i`, and value equal to that of `i` except that the bits are shifted `shift` places to the left (`-shift` places to the right if `shift` is negative). Zeros are shifted in from the other end. The argument `shift` must be an integer with value satisfying the inequality $|\texttt{shift}| \leq \texttt{bit\_size(i)}$.

ishftc (i, shift *[*, size*]*) returns an integer, with the same type parameter as `i`, and value equal to that of `i` except that the `size` rightmost bits (or all the bits if `size` is absent) are shifted circularly `shift` places to the left (`-shift` places to the right if `shift` is negative). The argument `shift` must be an integer with absolute value not exceeding the value of `size` (or `bit_size(i)` if `size` is absent).

shifta (i, shift) returns the bits of `i` shifted right by `shift` bits, but instead of shifting in zero bits from the left, the leftmost bit is replicated. The argument `shift` must be an integer with value satisfying the inequality $0 \leq \texttt{shift} \leq \texttt{bit\_size(i)}$.

shiftl (i, shift) returns the bits of `i` shifted left, equivalent to `ishft(i, shift)`. The argument `shift` must be an integer with value satisfying the inequality $0 \leq \texttt{shift} \leq \texttt{bit\_size(i)}$.

shiftr (i, shift) returns the bits of `i` shifted right, equivalent to `ishft(i, -shift)`. The argument `shift` must be an integer with value satisfying the inequality $0 \leq \texttt{shift} \leq \texttt{bit\_size(i)}$.

In each case, the `i` and `shift` are both of type integer (of any kind), `shift` must be in the range $0 \leq \texttt{shift} \leq \texttt{bit\_size(i)}$, and the result is of type integer with the same kind as `i`.

The advantages of `shiftl` and `shiftr` over `ishft` are:

- the shift direction is implied by the name, so one doesn't have to remember that a positive shift value means 'shift left' and a negative shift value means 'shift right';
- if the shift amount is variable, the code generated for shifting is theoretically more efficient (in practice, unless a lot of other things are being done to the values, the performance is going to be limited by the main memory bandwidth anyway, not the shift function).

9.10.5 Elemental subroutine

call mvbits (from, frompos, len, to, topos) copies the sequence of bits in `from` that starts at position `frompos` and has length `len` to `to`, starting at position

`topos`. The other bits of `to` are not altered. The arguments `from`, `frompos`, `len`, and `topos` are all integers with intent `in`, and they must have values that satisfy the inequalities: `frompos+len` $\leq$ `bit_size(from)`, `len` ≥ 0, `frompos` ≥ 0, `topos+len` $\leq$ `bit_size(to)`, and `topos` ≥ 0. The argument `to` is an integer with intent `inout`; it must have the same kind type parameter as `from`. The same variable may be specified for `from` and `to`.

9.10.6 Bitwise (unsigned) comparison

Four elemental functions are provided for performing bitwise comparisons, returning a default logical result. Bitwise comparisons treat integer values as **unsigned** integers; that is, the most significant bit is not treated as a sign bit but as having the value of 2^{b-1}, where b is the number of bits in the integer.

`bge (i, j)` returns the value true if `i` is bitwise greater than or equal to `j`, and the value false otherwise.

`bgt (i, j)` returns the value true if `i` is bitwise greater than `j`, and the value false otherwise.

`ble (i, j)` returns the value true if `i` is bitwise less than or equal to `j`, and the value false otherwise.

`blt (i, j)` returns the value true if `i` is bitwise less than `j`, and the value false otherwise.

The arguments `i` and `j` must either be of type integer or be 'boz' constants (Section 2.6.6); if of type integer, they need not have the same kind type parameter. For example, on a two's-complement processor with `integer(int8)` holding eight-bit integers, `-1_int8` has the bit pattern `z'ff'`, and this has the value 255 when treated as unsigned, so `bge(-1_int8, 255)` is true and `blt(-1_int8, 255)` is false.

9.10.7 Double-width shifting

Two unusual elemental functions provide double-width shifting. These functions concatenate `i` and `j` and shift the combined value left or right by `shift`; the result is the most significant half for a left shift and the least significant half for a right shift.

`dshiftl (i, j, shift)` returns the most significant half of a double-width left shift.

`dshiftr (i, j, shift)` returns the least significant half of double-width right shift.

One of the arguments `i` and `j` must be of type integer. The other is either of type integer with the same kind or is a 'boz' literal constant and is converted to the kind as if by the `int` function. The result is integer and of this kind. The `shift` argument must be an integer, but can be of any kind. For example, if `integer(int8)` holds eight-bit integers, `dshiftl(21_int8, 64_int8, 2)`, has the value `85_int8`.

In general, these functions are harder to understand and will perform worse than simply using ordinary shifts on integers of double the width, so they should be used only if the exact functionality is really what is required.

9.10.8 Bitwise reductions

Three transformational functions that perform bitwise reductions to reduce an integer array by one rank or to a scalar, `iall`, `iany`, and `iparity`, are described in Section 9.13.1.

9.10.9 Counting bits

Several elemental functions are provided for counting bits within an integer.

`leadz (i)` returns the number of leading (most significant) zero bits in `i`.

`popcnt (i)` returns the number of nonzero bits in `i`.

`poppar (i)` returns the value 1 if `popcnt(i)` is odd and 0 otherwise.

`trailz (i)` returns the number of trailing (least significant) zero bits in `i`.

The argument `i` may be any kind of integer, and the result is a default integer.

Note that the values of `popcnt`, `poppar`, and `trailz` depend only on the value of the argument, whereas the value of `leadz` depends also on the kind of the argument. For example, if `integer(int8)` holds eight-bit integers and `integer(int16)` holds 16-bit integers, `leadz(64_int8)` has the value 2, while `leadz(64_int16)` has the value 10; the values of `popcnt(64_k)`, `poppar(64_k)`, and `trailz(64_k)` are 1, 1, and 5, respectively, no matter what the kind value `k` is.

9.10.10 Producing bitmasks

Other elemental functions facilitate producing simple bitmasks:

`maskl (i [, kind])` returns an integer with the leftmost `i` bits set and the rest zero.

`maskr (i [, kind])` returns an integer with the rightmost `i` bits set and the rest zero.

The result type is integer with the specified kind (or default integer if no `kind` is specified). The argument `i` must be of type integer of any kind (the kind of `i` has no effect on the result), and with value in the range $0 \leq \mathtt{i} \leq b$, where b is the bit size of the result.

For example, if `integer(int8)` holds eight-bit integers, `maskl(3,int8)` is equal to `int(b'11100000',int8)` and `maskr(3,int8)` is equal to `7_int8`.

9.10.11 Merging bits

An elemental function merges bits from separate integers.

`merge_bits (i, j, mask)` returns the bits of `i` and `j` merged under the control of `mask`. The arguments `i`, `j`, and `mask` must be integers of the same kind or be 'boz' constants. At least one of `i` and `j` must be an integer, and a 'boz' constant is converted to that type as if by the `int` intrinsic; the result is of type integer with the same kind.

This function is modelled on the `merge` intrinsic, treating `1` and `0` bits as true and false, respectively. The value of the result is determined by taking the bit positions where `mask` is `1` from `i`, and the bit positions where `mask` is `0` from `j`; this is equal to `ior(iand(i, mask), iand(j, not(mask)))`.

9.11 Transfer function

The transfer function allows data of one type to be transferred to another without the physical representation being altered. This would be useful, for example, in writing a generic data storage and retrieval system. The system itself could be written for one type, default integer say, and other types handled by transfers to and from that type, for example:

```
integer          :: store
character(len=4) :: word        ! To be stored and retrieved
⋮
store = transfer(word, store)   ! Before storage
⋮
word  = transfer(store, word)   ! After retrieval
```

`transfer (source, mold [,size])` returns a result of type and type parameters those of `mold`. When `size` is absent, the result is scalar if `mold` is scalar, and it is of rank one and size just sufficient to hold all of `source` if `mold` is array valued. When `size` is present, the result is of rank one and size `size`. If the physical representation of the result is as long as or longer than that of `source`, the result contains `source` as its leading part and the value of the rest is processor dependent; otherwise the result is the leading part of `source`. As the rank of the result can depend on whether or not `size` is specified, the corresponding actual argument must not itself be an optional dummy argument.

9.12 Vector and matrix multiplication functions

There are two transformational functions that perform vector and matrix multiplications. They each have two arguments that are both of numeric type (integer, real, or complex) or both of logical type. The result is of the same type and type parameter as for the multiply or `and` operation between two such scalars. The functions `sum` and `any`, used in the definitions, are defined in Section 9.13.1.

`dot_product (vector_a, vector_b)` requires two arguments each of rank one and the same size. If `vector_a` is of type integer or type real, the function returns `sum(vector_a * vector_b)`; if `vector_a` is of type complex, it returns `sum(conjg(vector_a) * vector_b)`; and if `vector_a` is of type logical, it returns `any(vector_a .and. vector_b)`.

`matmul (matrix_a, matrix_b)` performs matrix multiplication. For numeric arguments, three cases are possible:

i) `matrix_a` has shape (n,m) and `matrix_b` has shape (m,k). The result has shape (n,k) and element (i,j) has the value `sum(matrix_a(`*i*`, :)*matrix_b(:,` *j*`))`.

ii) `matrix_a` has shape (*m*) and `matrix_b` has shape (*m, k*). The result has shape (*k*) and element (*j*) has the value `sum(matrix_a*matrix_b(:,` *j*`))`.

iii) `matrix_a` has shape (*n, m*) and `matrix_b` has shape (*m*). The result has shape (*n*) and element (*i*) has the value `sum(matrix_a(`*i*`, :)*matrix_b)`.

For logical arguments, the shapes are as for numeric arguments and the values are determined by replacing '`sum`' and '`*`' in the above expressions by '`any`' and '`.and.`'.

9.13 Transformational functions that reduce arrays

There are twelve transformational functions that perform operations on arrays such as summing their elements.

9.13.1 Single argument case

In their simplest form, these functions have a single array argument and return a scalar result. All except `count` have a result of the same type and type parameter as the argument. The mask array `mask`, used as an argument in `any`, `all`, `count`, and optionally in others, is also described in Section 7.7.

`all (mask)` returns the value true if all elements of the logical array `mask` are true or `mask` has size zero, and otherwise returns the value false.

`any (mask)` returns the value true if any of the elements of the logical array `mask` is true, and returns the value false if no elements are true or if `mask` has size zero.

`count (mask[, dim, kind])` returns the integer value that is the number of elements of the logical array `mask` that have the value true. The optional `dim` argument is described in the next section. The optional final `kind` argument must be a scalar integer constant expression. If present, it specifies the kind of the result, which is otherwise of default kind.

`iall (array)` returns an integer value in which each bit is 1 if all the corresponding bits of the elements of the integer array `array` are 1, and is 0 otherwise.

`iany (array)` returns an integer value in which each bit is 1 if any of the corresponding bits of the elements of the integer array `array` are 1, and is 0 otherwise.

`iparity (array)` returns an integer value in which each bit is 1 if the number of corresponding bits of the elements of the integer array `array` is odd, and is 0 otherwise.

`maxval (array)` returns the maximum value of an element of an integer, real, or character array (character comparison is explained in Section 3.5). If `array` has size zero, it returns the negative value of largest magnitude supported by the processor.

`minval (array)` returns the minimum value of an element of an integer, real, or character array (character comparison is explained in Section 3.5). If `array` has size zero, it returns the largest positive value supported by the processor.

`norm2 (x)` returns the L_2 norm of a real array `x`, that is, the square root of the sum of the squares of the elements. It returns the value zero if `array` has size zero. The standard recommends, but does not require, that `norm2` be calculated without undue overflow or underflow.

`parity (mask)` returns the value true if an odd number of the elements of the logical array `mask` are true, and false otherwise.

`product (array)` returns the product of the elements of an integer, real, or complex array. It returns the value one if `array` has size zero.

`sum (array)` returns the sum of the elements of an integer, real, or complex array. It returns the value zero if `array` has size zero.

9.13.2 Additional argument dim

Each of these functions except `count` has an alternative form with an additional second argument `dim`. The function `count` has an optional second argument `dim`. If `dim` is present, it must be a scalar integer and the operation is applied to all rank-one sections that span right through dimension `dim` to produce an array of rank reduced by one and extents equal to the extents in the other dimensions, or a scalar if the original rank is one. For example, if `a` is a real array of shape (4,5,6), `sum(a,dim=2)` is a real array of shape (4,6) and element (i, j) has value `sum(a(i,:,j))`. The actual argument corresponding to `dim` is not permitted to be an optional dummy argument of the procedure containing the call.[4]

Because the rank of the result of `count` depends on whether `dim` is specified (unless the original is rank one), the corresponding actual argument is not permitted itself to be an optional dummy argument, a disassociated pointer, or an unallocated allocatable object.

9.13.3 Optional argument mask

The functions `iall`, `iany`, `iparity`, `maxval`, `minval`, `product`, and `sum` have a third optional argument, a logical variable `mask`. If this is present, it must be conformable with the first argument and the operation is applied to the elements corresponding to true elements of `mask`; for example, `sum(a, mask = a>0)` sums the positive elements of the array `a`. The argument `mask` affects only the value of the function and does not affect the evaluation of arguments that are array expressions. The argument `mask` is permitted as the second positional argument when `dim` is absent.

[4]Except for `count`, this rule is relaxed in Fortran 2018, see Section 23.7.4.

9.14 Array inquiry functions

There are five functions for inquiries about the contiguity, bounds, shape, and size, of an array of any type. Because the result depends on only the array properties, the value of the array need not be defined.

9.14.1 Contiguity

Contiguity of the elements of an array can be tested.

`is_contiguous (array),` where `array` is an array of any type, returns a default logical scalar with the value `.true.` if `array` is contiguous, and `.false.` otherwise. If `array` is a pointer, it must be associated with a target.

9.14.2 Bounds, shape, and size

The following functions enquire about the bounds of an array. In the case of an allocatable array, it must be allocated; and in the case of a pointer, it must be associated with a target. An array section or an array expression is taken to have lower bounds 1 and upper bounds equal to the extents (like an assumed-shape array with no specified lower bounds). If a dimension has size zero, the lower bound is taken as 1 and the upper bound is taken as 0.

`lbound (array [,dim] )` when `dim` is absent, returns a rank-one default integer array holding the lower bounds. When `dim` is present, it must be a scalar integer and the result is a scalar default integer holding the lower bound in dimension `dim`. As the rank of the result depends on whether `dim` is specified, the corresponding actual argument must not itself be an optional dummy argument, a disassociated pointer, or an unallocated allocatable object.

`shape (source)` returns a rank-one default integer array holding the shape of the array or scalar `source`. In the case of a scalar, the result has size zero.

`size (array [,dim] )` returns a scalar default integer that is the size of the array `array` or extent along dimension `dim` if the scalar integer `dim` is present.

`ubound (array [,dim] )` is similar to `lbound` except that it returns upper bounds.

These functions have an optional `kind` argument at the end of the argument list. This argument specifies the kind of integer result the function returns. This is useful if a default integer is not big enough to contain the correct value (which may be the case on 64-bit machines). For example, in the code

```
real, allocatable :: a(:,:,:,:)
allocate (a(64,1024,1024,1024))
⋮
print *, size(a, kind=selected_int_kind(12))
```

the array `a` has a total of 2^{36} elements; on most machines this is bigger than `huge(0)`, so the `kind` argument is needed to get the right answer from the reference to the intrinsic function `size`.

9.15 Array construction and manipulation functions

There are eight functions that construct or manipulate arrays of any type.

9.15.1 The merge elemental function

`merge (tsource, fsource, mask)` is an elemental function. The argument `tsource` may have any type and `fsource` must have the same type and type parameters. The argument `mask` must be of type logical. The result is `tsource` if `mask` is true and `fsource` otherwise.

The principal application of `merge` is when the three arguments are arrays having the same shape, in which case `tsource` and `fsource` are merged under the control of `mask`. Note, however, that `tsource` or `fsource` may be scalar, in which case the elemental rules effectively broadcast it to an array of the correct shape.

9.15.2 Packing and unpacking arrays

The transformational function `pack` packs into a rank-one array those elements of an array that are selected by a logical array of conforming shape, and the transformational function `unpack` performs the reverse operation. The elements are taken in array element order.

`pack (array, mask [,vector] )`, when `vector` is absent, returns a rank-one array containing the elements of `array` corresponding to true elements of `mask` in array element order; `mask` may be scalar with value true, in which case all elements are selected. If `vector` is present, it must be a rank-one array of the same type and type parameters as `array` and size at least equal to the number t of selected elements; the result has size equal to the size n of `vector`; if $t < n$, elements i of the result for $i > t$ are the corresponding elements of `vector`.

`unpack (vector, mask, field)` returns an array of the type and type parameters of `vector` and shape of `mask`. The argument `mask` must be a logical array and `vector` must be a rank-one array of size at least the number of true elements of `mask`; `field` must be of the same type and type parameters as `vector` and must either be scalar or be of the same shape as `mask`. The element of the result corresponding to the ith true element of `mask`, in array element order, is the ith element of `vector`; all others are equal to the corresponding elements of `field` if it is an array or to `field` if it is a scalar.

9.15.3 Reshaping an array

The transformational function `reshape` allows the shape of an array to be changed, with possible permutation of the subscripts.

`reshape (source, shape [,pad] [,order] )` returns an array with shape given by the rank-one integer array `shape`, and type and type parameters those of the array `source`. The size of `shape` must be constant, and its elements must not be negative. If `pad` is present, it must be an array of the same type and type parameters as `source`. If `pad` is absent or of size zero, the size of the result must not exceed the size of `source`. If `order` is absent, the elements of the result, in array element order, are the elements of `source` in array element order followed by copies of `pad` in array element order. If `order` is present, it must be a rank-one integer array with a value that is a permutation of (1,2,...,*n*); the elements $\texttt{r}(s_1,\ldots,s_n)$ of the result, taken in subscript order for the array having elements $\texttt{r}(s_{\texttt{order}(1)},\ldots,s_{\texttt{order}(n)})$, are those of `source` in array element order followed by copies of `pad` in array element order. For example, if `order` has the value (3,1,2), the elements `r(1,1,1)`, `r(1,1,2)`, ..., `r(1,1,k)`, `r(2,1,1)`, `r(2,1,2)`, ... correspond to the elements of `source` and `pad` in array element order.

9.15.4 Transformational function for replication

`spread (source, dim, ncopies)` returns an array of type and type parameters those of `source` and of rank increased by one. The argument `source` may be scalar or array valued. The arguments `dim` and `ncopies` are integer scalars. The result contains `max(ncopies, 0)` copies of `source`, and element $(r_1,\ldots,r_{n+1})$ of the result is `source`$(s_1,\ldots,s_n)$ where $(s_1,\ldots,s_n)$ is $(r_1,\ldots,r_{n+1})$ with subscript `dim` omitted (or `source` itself if it is scalar).

9.15.5 Array shifting functions

`cshift (array, shift [,dim] )` returns an array of the same type, type parameters, and shape as `array`. The argument `shift` is of type integer and must be scalar if `array` is of rank one. If `shift` is scalar, the result is obtained by shifting every rank-one section that extends across dimension `dim` circularly `shift` times. The argument `dim` is an integer scalar and, if it is omitted, it is as if it were present with the value 1. The direction of the shift depends on the sign of `shift`, being to the left for a positive value and to the right for a negative value. Thus, for the case with `shift=1` and `array` of rank one and size *m*, the element *i* of the result is `array(i+1)`, where $i = 1,2,\ldots,m-1$, and element *m* is `array(1)`. If `shift` is an array, it must have the same shape as that of `array` with dimension `dim` omitted, and it supplies a separate value for each shift. For example, if `array` is of rank three and shape (k,l,m) and `dim` has the value 2, `shift` must be of shape (k,m) and supplies a shift for each of the $k \times m$ rank-one sections in the second dimension of `array`.

`eoshift (array, shift [,boundary] [,dim] )` is identical to `cshift` except that an end-off shift is performed and boundary values are inserted into the gaps

so created. The argument `boundary` may be omitted when `array` has intrinsic type, in which case the value zero is inserted for the integer, real, and complex cases; false in the logical case; and blanks in the character case. If `boundary` is present, it must have the same type and type parameters as `array`; it may be scalar and supply all needed values or it may be an array whose shape is that of `array` with dimension `dim` omitted and supply a separate value for each shift.

9.15.6 Matrix transpose

The `transpose` function performs a matrix transpose for any array of rank two.

`transpose (matrix)` returns an array of the same type and type parameters as the rank-two array `matrix`. If `matrix` has shape $[m,n]$, the result has shape $[n,m]$. If `matrix` has lower bounds $lb1$ and $lb2$ (returned by `lbound(matrix,1)` and `lbound(matrix,2)`), element (i, j) of the result is `matrix(`$j+lb1-1$`,` $i+lb2-1$`)`.

9.16 Transformational functions for geometric location

There are two transformational functions that find the locations of the maximum and minimum values of an integer, real, or character array. They each have an optional `kind` argument that must be a scalar integer constant expression and controls the kind of the result, which otherwise is default integer.

Also, they have a final, optional `back` argument to indicate whether the first or last occurrence is desired; it must be a scalar logical. For example, `maxloc([1,4,4,1])` is equal to 2, whereas `maxloc([1,4,4,1], back=.true.)` is equal to 3.

`maxloc (array [, mask] [, kind] [, back])` returns a rank-one integer array of size equal to the rank of `array`. Its value is the sequence of subscripts of an element of maximum value (among those corresponding to true values of the conforming logical variable `mask` if it is present), as though all the declared lower bounds of `array` were 1. If there is more than one such element, the first in array element order is taken. If there are no elements, the result has all elements zero.

`maxloc (array, dim [, mask] [, kind] [, back])` returns an integer array of shape equal to that of `array` with dimension `dim` omitted, where `dim` is a scalar integer with value in the range $1 \leq$ `dim` $\leq$ rank(`array`), or a scalar if the original rank is one. The value of each element of the result is the position of the first element of maximum value in the corresponding rank-one section spanning dimension `dim`, among those elements corresponding to true values of the conforming logical variable `mask` when it is present. If there are no elements, the result has all elements zero.

`minloc (array [, mask] [, kind] [, back])` is identical to `maxloc (array [, mask] [, kind] [, back])` except that the position of an element of minimum value is obtained.

`minloc (array, dim [,mask] [, kind] [, back])` is identical to `maxloc (array, dim [,mask] [, kind] [, back])` except that positions of elements of minimum value are obtained.

For an array of any intrinsic type, the transformational intrinsic function `findloc` returns the position of an element with the specified value, or zero if no such element is found. Its form is as follows:

`findloc (array, value [, mask] [, kind] [, back])`
searches the whole of `array`, possibly masked by `mask`, for `value`, and returns the vector of subscript positions identifying that element. The `array` argument must be of intrinsic type, and `value` must be a scalar of comparable type and kind (not necessarily the same type). If present, `mask` must be of type logical and conformable with `array`, `kind` must be a scalar integer constant expression, and `back` must be a scalar logical. If `back` is present and true, the function finds the last suitable value in `array`, otherwise it finds the first such value.

`findloc (array, value, dim [, mask] [, kind] [, back])`
reduces dimension `dim` of `array`, the result being the position in each vector along dimension `dim` where the element was found. The arguments are the same as before, except for `dim` which must be a scalar integer.

The result type is integer with the specified kind (or default integer if no `kind` is specified). For example, `findloc([(i, i = 10, 1000, 10)], 470)` has the value `47`.

Note that because we are searching for equality, any intrinsic type may be used (whereas `maxloc` and `minloc` do not allow complex or logical); for type logical, the `.eqv.` operation is used for the comparison.

9.17 Transformational function for disassociated or unallocated

The function `null` is available to give the disassociated status to a pointer or unallocated status to an allocatable entity.

`null ([mold])` returns a disassociated pointer or an unallocated allocatable entity.
The argument `mold` is a pointer or allocatable. The type, type parameter, and rank of the result are those of `mold` if it is present and otherwise are those of the object with which it is associated. In an actual argument associated with a dummy argument of assumed character length, `mold` must be present.

9.18 Non-elemental intrinsic subroutines

There are also in Fortran non-elemental intrinsic subroutines, which were chosen to be subroutines rather than functions because of the need to return information through the arguments.

9.18.1 Real-time clock

There are two subroutines that return information from the real-time clock, the first based on the ISO 8601 (Representation of dates and times) standard. It is assumed that there is a basic system clock that is incremented by one for each clock count until a maximum `count_max` is

reached and on the next count is set to zero. Default values are returned on systems without a clock. All the arguments have intent `out`.

`call date_and_time (`*`[`*`date`*`][,time`*`][,zone`*`][,values`*`])` returns the following (with default values blank or `-huge(0)`, as appropriate, when there is no clock).

`date` is a scalar character variable. It is assigned the value of the century, year, month, and day in the form *ccyymmdd*.

`time` is a scalar character variable. It is assigned the value of the time as hours, minutes, seconds, and milliseconds in the form *hhmmss.sss*.

`zone` is a scalar character variable. It is assigned the value of the difference between local time and UTC (also known as Greenwich Mean Time) in the form *Shhmm*, corresponding to sign, hours, and minutes. For example, a processor in New York in winter would return the value `-0500`.

`values` is a rank-one default integer array of size at least 8 holding the sequence of values: the year, the month of the year, the day of the month, the time difference in minutes with respect to UTC, the hour of the day, the minutes of the hour, the seconds of the minute, and the milliseconds of the second.

`call system_clock (`*`[`*`count`*`][,count_rate`*`][,count_max`*`])` returns the following.

`count` is a scalar integer[5] holding a processor-dependent value based on the current value of the processor clock, or `-huge(0)` if there is no clock. On the first call, the processor may set an initial value that may be zero.

`count_rate` is a scalar integer[5] or real[6] holding an approximation to the number of clock counts per second, or zero if there is no clock.

`count_max` is a scalar integer[5] holding the maximum value that `count` may take, or zero if there is no clock.

9.18.2 CPU time

There is a non-elemental intrinsic subroutine that returns the processor time.

`call cpu_time (time)` returns the following:

`time` is a scalar real that is assigned a processor-dependent approximation to the processor time in seconds, or a processor-dependent negative value if there is no clock.

[5] It may be of any kind in order to accommodate systems with a clock rate that is too high to be represented in a default integer.

[6] This is to provide a more accurate result on systems whose clock does not tick an integral number of times each second.

The exact definition of time is left imprecise because of the variability in what different processors are able to provide. The primary purpose is to compare different algorithms on the same computer or discover which parts of a calculation on a computer are the most expensive. Sections of code can be timed, as in the example

```
real :: t1, t2
⋮
call cpu_time(t1)
⋮                    ! Code to be timed.
call cpu_time(t2)
write (*,*) 'Time taken by code was ', t2-t1, ' seconds'
```

9.18.3 Random numbers

A sequence of pseudorandom numbers is generated from a seed that is held as a rank-one array of integers. The subroutine `random_number` returns the pseudorandom numbers and the subroutine `random_seed` allows an inquiry to be made about the size or value of the seed array, and the seed to be reset. The subroutines provide a portable interface to a processor-dependent sequence.

`call random_number (harvest)` returns a pseudorandom number from the uniform distribution over the range $0 \leq x < 1$ or an array of such numbers; `harvest` has intent `out`, may be a scalar or an array, and must be of type real.

`call random_seed (`*`[size]  [put]  [get]`*` )` has the following arguments:

`size` has intent `out` and is a scalar default integer that the processor sets to the size n of the seed array.

`put` has intent `in` and is a default integer array of rank one and size n that is used by the processor to reset the seed. A processor may set the same seed value for more than one value of `put`.

`get` has intent `out` and is a default integer array of rank one and size n that the processor sets to the current value of the seed. This value can be used later as `put` to replay the sequence from that point, or in a subsequent program execution to continue from that point.

No more than one argument may be specified; if no argument is specified, the seed is set to a processor-dependent value. Thus, this value may be identical for each call, or different.

9.18.4 Executing another program

The ability to execute another program from within a Fortran program is provided by the intrinsic subroutine `execute_command_line`; as its name suggests, this passes a 'command line' to the processor which will interpret it in a totally system-dependent manner. For example,

```
call execute_command_line('ls -l')
```

is likely to produce a directory listing on Unix and an error message on Windows. The full syntax is as follows:

call execute_command_line (command*[*, **wait*]*[*, **exitstat***]*[*, **cmdstat***]*[*, **cmdmsg***]*)** where the arguments are as follows:

command has intent `in` and is a scalar default character string containing the command line to be interpreted by the processor.

wait has intent `in` and is a scalar default logical indicating whether the command should be executed asynchronously (`wait=.false.`), or whether the procedure should wait for it to terminate before returning to the Fortran program (the default).

exitstat has intent `inout` and is a scalar default integer variable that, unless `wait` is false, will be assigned the 'process exit status' from the command (the meaning of this is also system dependent).

cmdstat has intent `out` and is a scalar default integer variable that is assigned zero if `execute_command_line` itself executed without error, -1 if the processor does not support command execution, -2 if `wait=.true.` was specified but the processor does not support asynchronous command execution, and a positive value if any other error occurred.

cmdmsg has intent `inout` and is a scalar default character string to which, if `cmdstat` is assigned a positive value, is assigned an explanatory message.

If any error occurs (such that a nonzero value would be assigned to `cmdstat`) and `cmdstat` is not present, the program is error-terminated.

Note that even if the processor supports asynchronous command execution, there is no mechanism provided for finding out later whether the command being executed asynchronously has terminated or what its exit status was.

9.19 Access to the computing environment

Intrinsic procedures are available to provide information about environment variables, and about the command by which a program is executed.

9.19.1 Environment variables

Most operating systems have some concept of an *environment variable*, associating names with values. Access to these is provided by an intrinsic subroutine.

call get_environment_variable (name*[*, **value*][*, **length***][*, **status***]*[*, **trim_name***]*)** where the arguments are defined as follows:

name has intent `in` and is a scalar default character string containing the name of the environment variable to be retrieved. Trailing blanks are not significant unless `trim_name` is present and false. Case may or may not be significant.

value has intent `out` and is a scalar default character variable; it receives the value of the environment variable (truncated or padded with blanks if the `value` argument is shorter or longer than the environment variable's value). If there is no such variable, there is such a variable but it has no value, or the processor does not support environment variables, this argument is set to blanks.

length has intent `out` and is a scalar default integer variable; if the specified environment variable exists and has a value, the `length` argument is set to the length of that value, otherwise it is set to zero.

status has intent `out` and is a scalar default integer; it receives the value 1 if the environment variable does not exist, 2 if the processor does not support environment variables, a number greater than 2 if an error occurs, −1 if the `value` argument is present but too short, and zero otherwise (indicating that no error or warning condition has occurred).

trim_name has intent `in` and is a scalar of type logical; if this is false, trailing blanks in `name` will be considered significant if the processor allows environment variable names to contain trailing blanks.

9.19.2 Information about the program invocation

Two different methods of retrieving information about the command are provided, reflecting the two approaches in common use.

The Unix-like method is provided by two procedures: a function which returns the number of command arguments and a subroutine which returns an individual argument.[7] These are:

command_argument_count () returns, as a scalar default integer, the number of command arguments. If the result is zero, either there were no arguments or the processor does not support the facility. If the command name is available as an argument, it is not included in this count.

call get_command_argument (number*[*, value*]* *[*, length*]* *[*, status*]*)
where the arguments are defined as follows:

number has intent `in` and is a scalar default integer indicating the number of the argument to return. If the command name is available as an argument, it is number zero.

value has intent `out` and is a scalar default character variable; it receives the value of the indicated argument (truncated or padded with blanks if the character variable is shorter or longer than the command argument).

length has intent `out` and is a scalar default integer variable; it receives the length of the indicated argument.

status has intent `out` and is a scalar default integer variable; it receives a positive value if that argument cannot be retrieved, −1 to indicate that the `value` variable was shorter than the command argument, and zero otherwise.

[7] All these three procedures are enhanced in Fortran 2018, see Section 23.7.6.

The other paradigm for command processing provides a simple command line, not broken up into arguments. This is retrieved by the intrinsic subroutine

`call get_command (`*`[`***` command `***`] [ , `***` length `***`] [, `***` status `***`])`** where

`command` has intent `out` and is a scalar default character variable; it receives the value of the command line (truncated or padded with blanks if the variable is shorter or longer than the actual command line).

`length` has intent `out` and is a scalar default integer variable; it receives the length of the actual command line, or zero if the length cannot be determined.

`status` has intent `out` and is a scalar default integer variable; it receives a positive value if the command line cannot be retrieved, -1 if `command` was present but the variable was shorter than the length of the actual command line, and zero otherwise.

9.20 Elemental functions for I/O status testing

Two elemental intrinsic functions are provided for testing the I/O status value returned through the `iostat=` specifier (Section 10.7). Both functions accept an argument of type integer, and return a default logical result.

`is_iostat_end(i)` returns the value true if `i` is an I/O status value that corresponds to an end-of-file condition, and false otherwise.

`is_iostat_eor(i)` returns the value true if `i` is an I/O status value that corresponds to an end-of-record condition, and false otherwise.

9.21 Size of an object in memory

`storage_size (a` ***`[`*** **`, kind`** ***`]`*** **`)`** returns the size, in bits, that would be taken in memory by an array element with the dynamic type and type parameters (Section 15.3.2) of `a`.

The argument `a` may be of any type or rank (including a scalar). It is permitted to be an undefined pointer unless it is polymorphic, and is permitted to be a disassociated pointer or unallocated allocatable unless it has a deferred type parameter or is unlimited polymorphic.

The return type is integer with the specified kind, or default kind if `kind` is not present.

Note that the standard does not require the same size for named variables, array elements, and structure components of the same type; indeed, frequently these will have different padding to improve memory address alignment and thus performance.

Furthermore, if `a` is of a derived type with allocatable components or components whose size depends on the value of a length type parameter, the compiler is allowed to store those components separately from the rest of the variable, with a descriptor in the variable pointing to the additional storage. It is unclear whether `storage_size` will include the space taken up by such components, especially in the length type parameter case. Therefore, use of this function should be avoided for such problematic cases.

9.22 Miscellaneous procedures

The following procedures, introduced elsewhere, are fully described in the sections listed:[8]

- the subroutine `move_alloc`, Section 6.8;
- the inquiry functions `extends_type_of` and `same_type_as`, Section 15.13;
- the inquiry functions `image_index`, `lcobound`, and `ucobound`, Section 17.16.1;
- the transformational functions `num_images` and `this_image`, Section 17.16.2;
- the atomic subroutines `atomic_define` and `atomic_ref`, Section A.9.1; and
- the double-precision functions `dble` and `dprod`, Section A.6.

9.23 Intrinsic modules

An intrinsic module is one that is provided by the Fortran processor instead of the user or a third party. A Fortran processor provides at least five intrinsic modules: `iso_fortran_env`, `ieee_arithmetic`, `ieee_exceptions`, `ieee_features`, and `iso_c_binding`, and may provide additional intrinsic modules.

The intrinsic module `iso_fortran_env` provides information about the Fortran environment, and is described in the next section. The IEEE modules provide access to facilities from the IEEE arithmetic standard and are described in Chapter 18. The intrinsic module `iso_c_binding` provides support for interoperability with C and is described in Chapter 19.

It is possible for a program to use an intrinsic module and a user-defined module of the same name, though they cannot both be referenced from the same scoping unit. To use an intrinsic module in preference to a user-defined one of the same name, the `intrinsic` keyword is specified on the `use` statement, for example

```
use, intrinsic :: ieee_arithmetic
```

Similarly, to ensure that a user-defined module is accessed in preference to an intrinsic module, the `non_intrinsic` keyword is used, for example:

```
use, non_intrinsic :: random_numbers
```

If both an intrinsic module and a user-defined module are available with the same name, a `use` statement without either of these keywords accesses the user-defined module. However, should the compiler not be able to find the user's module it would access the intrinsic one instead without warning; therefore we recommend that programmers avoid using the same name for a user-defined module as that of a known intrinsic module (or that the `non_intrinsic` keyword be used).

Intrinsic modules should always be used with an `only` clause (Section 8.14), as vendors or future standards could make additions to the module.

[8] Further procedures, introduced in Fortran 2018, are described in Sections 23.6 and 23.16.

9.24 Fortran environment

The intrinsic module `iso_fortran_env` provides information about the Fortran environment.

9.24.1 Named constants

The following named constants, which are scalars of type default integer, are useful in the contexts of data transfer (Chapter 10), external files (Chapter 12), and coarray processing (Chapter 17).

`character_storage_size` The size in bits of a character storage unit, Section A.2.

`error_unit` The unit number for a preconnected output unit suitable for reporting errors.

`file_storage_size` The size in bits of a file storage unit (the unit of measurement for the record length of an external file, as used in the `recl=` clause of an `open` or `inquire` statement).

`input_unit` The unit number for the preconnected standard input unit (the same one that is used by `read` without a unit number, or with a unit specifier of `*`).

`iostat_end` The value returned by `iostat=` to indicate an end-of-file condition.

`iostat_eor` The value returned by `iostat=` to indicate an end-of-record condition.

`iostat_inquire_internal_unit` The value returned in an `iostat=` specifier by an `inquire` statement to indicate that a file unit number identifies an internal unit.

`numeric_storage_size` The size in bits of a numeric storage unit, Section A.2.

`stat_locked` The value returned by a `stat=` variable in a `lock` statement for a variable locked by the executing image, Section 17.13.3.

`stat_locked_other_image` The value returned by a `stat=` variable in a `lock` statement for a variable locked by another image, Section 17.13.3.

`stat_unlocked` The value returned by a `stat=` variable in a `unlock` statement for an unlocked variable, Section 17.13.3.

`stat_stopped_image` The value returned by a `stat=` variable to indicate a stopped image, Section 17.13.6.

`output_unit` The unit number for the preconnected standard output unit (the same one that is used by `print`, or by `write` with a unit specifier of `*`).

Unlike normal unit numbers, the special unit numbers might be negative, but they will not be -1 (this is because -1 is used by the `number=` clause of the `inquire` statement to mean that there is no unit number). The error reporting unit `error_unit` might be the same as the standard output unit `output_unit`.

9.24.2 Compilation information

Two inquiry functions are available in the module `iso_fortran_env` to return information about the compiler (the so-called program translation phase).

`compiler_version ()` returns a string describing the name and version of the compiler used.

`compiler_options ()` returns a string describing the options used during compilation.

In each case the string is a default character scalar.

These functions may be used in initialization expressions, for example

```
module my_module
  use iso_fortran_env, only: compiler_options, compiler_version
  private compiler_options, compiler_version
  character(*), parameter :: compiled_by = compiler_version()
  character(*), parameter :: compiled_with = compiler_options()
  :
end module
```

There are no actual requirements on the length of these strings or on their contents, but it is expected that they will contain something useful and informative. For example `compiler_version()` could return the string `'NAG Fortran 6.0(1273)'`, and `compiler_options()` could return the string `'-C=array -O3'`.

9.24.3 Names for common kinds

Named constants for some kind values for integer and real types are available in the module `iso_fortran_env`; these are the default integer scalars:

`int8`	8-bit integer
`int16`	16-bit integer
`int32`	32-bit integer
`int64`	64-bit integer
`real32`	32-bit real
`real64`	64-bit real
`real128`	128-bit real

For example, in the code in Figure 9.1 the use of `int64` allows the subroutine to process very large arrays.

If the compiler supports more than one kind with a particular size, the standard does not specify which one will be chosen for the constant. If the compiler does not support a kind with a particular size, that constant will have a value of -2 if it supports a kind with a larger size, and -1 if it does not support any larger size.

This can be used together with `merge` to specify a desired size with a fallback onto other predetermined sizes if that one is not available, as shown in Figure 9.2.

Figure 9.1 Use of `int64`.

```
subroutine process(array)
  use iso_fortran_env
  real array(:, :)
  integer(int64) i, j
  do j=1, ubound(array, 2, int64)
    do i=1, ubound(array, 1, int64)
      ⋮ ! do something with array(i, j)
    end do
  end do
end subroutine
```

Figure 9.2 Kind selection with standard named constants.

```
subroutine process_bytes(bytes)
  use iso_fortran_env
  integer(merge(int8, merge(int16, int32, int16>=0), int8>=0)) bytes
!
  if (kind(bytes)==int8) then
    ⋮ ! process 8-bit bytes
  else if (kind(bytes)==int16) then
    ⋮ ! process 8-bit bytes in pairs
  else
    ⋮ ! process quadruples of 8-bit bytes
  end if
end subroutine
```

The named constants `atomic_int_kind` and `atomic_logical_kind` are default integer scalars available in the module `iso_fortran_env` for the kind values for integer and logical variables used as arguments for the intrinsic subroutines `atomic_define` and `atomic_ref`, Section A.9.1.

9.24.4 Kind arrays

Named array constants containing all the kind type parameter values for the intrinsic types that are supported by the processor are available in the intrinsic module `iso_fortran_env`. The named constants `character_kinds`, `integer_kinds`, `logical_kinds`, and `real_kinds` contain the supported kinds of type character, integer, logical, and real, respectively. These arrays are of type default integer, and have a lower bound of 1. The order of values in each array is processor dependent.

9.24.5 Lock type

The derived type `lock_type` is defined in the intrinsic module `iso_fortran_env`. It is described in Section 17.13.3.

9.25 Summary

In this chapter we introduced the four categories of intrinsic procedures, explained the `intrinsic` statement, and gave detailed descriptions of all the procedures. Intrinsic modules have also been introduced.

Exercises

1. Write a program to calculate the real roots or pairs of complex-conjugate roots of the quadratic equation $ax^2 + bx + c = 0$ for any real values of a, b, and c. The program should read these three values and print the results. Use should be made of the appropriate intrinsic functions.
2. Repeat Exercise 1 of Chapter 5, avoiding the use of `do` constructs.
3. Given the rules explained in Sections 3.13 and 9.2, what are the values printed by the following program?

```
program main
   real, target  :: a(3:10)
   real, pointer :: p1(:), p2(:)
!
   p1 => a(3:9:2)
   p2 => a(9:3:-2)
   print *, associated(p1, p2)
   print *, associated(p1, p2(4:1:-1))
end program main
```

4. In the following program, two pointer assignments, one to an array and the other to an array section, are followed by a subroutine call. Bearing in mind the rules given in Sections 3.13, 5.7.1, and 9.14.2, what values does the program print?

```
program main
   real, target  :: a(5:10)
   real, pointer :: p1(:), p2(:)
   p1 => a
   p2 => a(:)
   print *, lbound (a), lbound (a(:))
   print *, lbound (p1), lbound (p2)
   call what (a, a(:))
contains
   subroutine what (x, y)
      real, intent (in) :: x(:), y(:)
      print *, lbound (x), lbound (y)
   end subroutine what
end program main
```

5. Write a program that displays the sum of all the numbers on its command line.

6. Write a module that implements the standard `random_number` interface, for single and double precision real numbers, by the 'good, minimal standard' generator from *Random Number Generators: Good Ones Are Hard to Find* (S. K. Park and K. W. Miller, CACM October 1988, Volume 31 Number 10, pp 1192−1201). This is a parametric multiplicative linear congruential algorithm:

$$x_{new} = \mathrm{mod}(16807 x_{old}, 2^{31} - 1).$$

10. Data transfer

10.1 Introduction

Fortran has, in comparison with many other high-level programming languages, a particularly rich set of facilities for input/output (I/O), but it is an area of Fortran into which not all programmers need to delve very deeply. For most small-scale programs it is sufficient to know how to read a few data records containing input variables, and how to transmit to a screen or printer the results of a calculation. In large-scale data processing, on the other hand, the programs often have to deal with huge streams of data to and from many files; in these cases it is essential that great attention be paid to the way in which the I/O is designed and coded, as otherwise both the execution time and the real time spent in the program can suffer dramatically. The term **external file** is used for a collection of data outside the main memory. A file is usually organized into a sequence of **records**, with access either sequential or direct. An alternative is provided by stream access, discussed in Section 10.16.

This chapter begins by discussing the various forms of **formatted I/O**, that is, I/O which deals with records that do not use the internal number representation of the computer but rather a character string that can be displayed. It is also the form usually needed for transmitting data between different kinds of computers. The so-called **edit descriptors**, which are used to control the translation between the internal number representation and the external format, are then explained. Finally, the topics of unformatted (or binary) I/O and direct-access files are covered.

10.2 Number conversion

The ways in which numbers are stored internally by a computer are the concern of neither the Fortran standard nor this book. However, if we wish to output values – to display them on a screen or to print them – then their internal representations must be converted into a character string that can be read in a normal way. For instance, the contents of a given computer word may be (in hexadecimal) 1d7dbf and correspond to the value -0.000450. For our particular purpose, we may wish to display this quantity as `-.000450`, or as `-4.5E-04`, or rounded to one significant digit as `-5E-04`. The conversion from the internal representation to the external form is carried out according to the information specified by an edit descriptor contained in a *format specification*. These will both be dealt with fully in the next chapter; for the moment, it is sufficient to give a few examples. For instance, to print an integer value in a field ten characters wide, we would use the edit descriptor `i10`, where `i` stands for integer

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

conversion and `10` specifies the width of the output field. To print a real quantity in a field of ten characters, five of which are reserved for the fractional part of the number, we specify `f10.5`. The edit descriptor `f` stands for floating-point (real) conversion, `10` is the total width of the output field, and `5` is the width of the fractional part of the field. If the number given above were to be converted according to this edit descriptor, it would appear as *bb*`-0.00045`, where *b* represents a blank. To print a character variable in a field of ten characters we would specify `a10`, where `a` stands for alphanumeric conversion.

A format specification consists of a list of edit descriptors enclosed in parentheses, and can be coded either as a default character expression, for instance

```
'(i10, f10.3, a10)'
```

or as a separate `format` statement, referenced by a statement label, for example

```
10 format(i10, f10.3, a10)
```

To print the scalar variables `j`, `b`, and `c`, of types integer, real, and character, respectively, we may then write either

```
print '(i10, f10.3, a10)', j,b,c
```

or

```
    print 10, j,b,c
10 format(i10, f10.3, a10)
```

The first form is normally used when there is only a single reference in a scoping unit to a given format specification, and the second when there are several or when the format is complicated. The part of the statement designating the quantities to be printed is known as the *output list* and forms the subject of the following section.

10.3 I/O lists

The quantities to be read or written by a program are specified in an I/O list. For output they may be expressions, but for input they must be variables. In both cases, list items may be implied-`do` lists of quantities. Examples are shown in Figure 10.1, where we note the use of a *repeat count* in front of those edit descriptors that are required repeatedly. A repeat count must be a positive integer literal constant and must not have a kind type parameter. Function references are permitted in an I/O list (but subject to conditions described in Section 11.7).

In all these examples, except the last one, the expressions consist of single variables and would be equally valid in input statements using the `read` statement, for example

```
read '(i10)', i
```

Such statements may be used to read values which are then assigned to the variables in the input list.

If an array appears as an item, it is treated as if the elements were specified in array element order. For example, the third of the `print` statements in Figure 10.1 could have been written

```
print '(3f10.3)', a(1:3)
```

Figure 10.1 Examples of formatted output.

```
integer             :: i
real, dimension(10) :: a
character(len=20)   :: word
print '(i10)',     i
print '(10f10.3)', a
print '(3f10.3)',  a(1),a(2),a(3)
print '(a10)',     word(5:14)
print '(5f10.3)', (a(i), i=1,9,2)
print '(2f10.3)',  a(1)*a(2)+i, sqrt(a(3))
```

Figure 10.2 An invalid input item (array element appears twice).

```
integer :: j(10), k(3)
⋮
k = (/ 1, 2, 1 /)
read '(3i10)', j(k)     ! Invalid because j(1) appears twice
```

However, no element of the array may appear more than once in an input item. Thus, the case in Figure 10.2 is not allowed.

If an allocatable array appears as an item, it must be allocated.

Any pointer in an I/O list must be associated with a target, and transfer takes place between the file and the target.

For formatted I/O,[1] unless defined input/output (Section 11.6) is in use, an item of derived type with no allocatable or pointer components at any level of component selection is treated as if the components were specified in the same order as in the type declaration. This rule is applied repeatedly for components of derived type, so that it is as if we specified the list of items of intrinsic type that constitute its ultimate components. For example, if `p` and `t` are of the types `point` and `triangle` of Figure 2.1, the statement

```
read '(8f10.5)', p, t
```

has the same effect as the statement

```
read '(8f10.5)', p%x, p%y, t%a%x, t%a%y, t%b%x,       &
                           t%b%y, t%c%x, t%c%y
```

During this expansion each component must be accessible (it may not, for example, be a `private` component of a `public` type).

An object in an I/O list is not permitted to be of a derived type that has an allocatable or pointer component at any level of component selection, unless that component is part of an item that would be processed by defined input/output. One reason for this restriction is that an I/O list item that is allocatable or a pointer is required to be allocated or associated, so any unallocated or disassociated component would be problematic. Another is that these

[1] Unformatted I/O handling of derived types is described in Section 10.12.

components usually vary in size dynamically, which could lead to ambiguous or confusing I/O. Finally, with a recursive data structure using pointers, either a disassociated pointer would be reached, or there would be a circular chain of pointers resulting in infinite I/O. Such types are better handled by doing the I/O individually for individual components with a procedure, or by using the defined input/output feature (see Section 11.6).

An I/O list may include an implied-`do` list, as illustrated by the fifth `print` statement in Figure 10.1. The general form is

(*do-object-list*, *do-var* = *expr*, *expr* [, *expr*])

where each *do-object* is a variable (for input), an expression (for output), or is itself an implied-`do` list; *do-var* is a named scalar integer variable; and each *expr* is a scalar integer expression. The loop initialization and execution is the same as for a (possibly nested) set of `do` constructs (Section 4.4). In an input list, a variable that is an item in a *do-object-list* must not be a *do-var* of any implied-`do` list in which it is contained, nor be associated[2] with such a *do-var*. In an input or output list, no *do-var* may be a *do-var* of any implied-`do` list in which it is contained or be associated with such a *do-var*.

Note that a zero-sized array, or an implied-`do` list with a zero iteration count, may occur as an item in an I/O list. Such an item corresponds to no actual data transfer.

10.4 Format definition

In the `print` and `read` statements of the previous section, the format specification was given each time in the form of a character constant immediately following the keyword. In fact, there are three ways in which a format specification may be given. They are as follows.

A default character expression whose value commences with a format specification in parentheses:

```
print '(f10.3)', q
```

or

```
character(len=*), parameter :: form='(f10.3)'
⋮
print form, q
```

or

```
character :: carray(7) = (/ '(','f','1','0','.','3',')' /)
⋮
print carray, q ! Elements of an array are concatenated.
```

or

[2] Such an association could be established by pointer association.

```
character(4) :: carr1(10)
character(3) :: carr2(10)
integer      :: i, j
⋮
carr1(10) = '(f10'
carr2(3) = '.3)'
⋮
i = 10
j = 3
⋮
print carr1(i)//carr2(j), q
```

From these examples it may be seen that it is possible to program formats in a flexible way, and particularly that it is possible to use arrays, expressions, and also substrings in a way which allows a given format to be built up dynamically at execution time from various components. Any character data that might follow the trailing right parenthesis are ignored and may be undefined. In the case of an array, its elements are concatenated in array element order. However, on input *no* component of the format specification may appear also in the input list, or be associated with it. This is because the standard requires that the whole format specification be established *before* any I/O takes place. Further, no redefinition or undefinition of any characters of the format is permitted during the execution of the I/O statement.

An asterisk This is a type of I/O known as **list-directed I/O**, in which the format is defined by the computer system at the moment the statement is executed, depending on both the type and magnitude of the entities involved. This facility is particularly useful for the input and output of small quantities of values, especially in temporary code which is used for test purposes, and which is removed from the final version of the program:

```
print *, 'Square-root of q = ', sqrt(q)
```

This example outputs a character constant describing the expression which is to be output, followed by the value of the expression under investigation. On the screen, this might appear as

```
Square-root of q = 4.392246
```

the exact format being dependent on the computer system used. Character strings in this form of output are normally undelimited, as if an `a` edit descriptor were in use, but an option in the `write` and `open` statements (Sections 10.18 and 12.4) may be used to require that they be delimited by apostrophes or quotation marks. Complex constants are represented as two real values separated by a comma and enclosed in parentheses. Logical variables are represented as `T` for true and `F` for false. Except for adjacent undelimited strings, values are separated by spaces or commas. The processor may represent a sequence of r identical values c by the form r`*`c. Further details of list-directed input/output are deferred until Section 10.9.

A statement label referring to a `format` statement containing the relevant specification between parentheses:

```
      print 100, q
      ⋮
  100 format(f10.3)
```

The `format` statement must appear in the same scoping unit, before the `contains` statement if it has one. It is customary either to place each format statement immediately after the first statement which references it, or to group them all together just before the `contains` or `end` statement. It is also customary to have a separate sequence of numbers for the statement labels used for `format` statements. A given `format` statement may be used by any number of formatted I/O statements, whether for input or for output.

Blank characters may precede the left parenthesis of a format specification, and may appear at any point within a format specification with no effect on the interpretation, except within a character string edit descriptor (Section 11.2).

10.5 Unit numbers

Input/output operations are used to transfer data between the variables of an executing program, as stored in the computer, and an external medium. There are many types of external media: the screen, printer, hard disk, memory stick, and DVD are perhaps the most familiar. Whatever the device, a Fortran program regards each one from which it reads or to which it writes as a **unit**, and each unit, with two exceptions, has associated with it a **unit number**. This number must not be negative. Thus, we might associate with a DVD from which we are reading the unit number 10, and with a hard disk to which we are writing the unit number 11. All program units of an executable program that refer to a particular unit number are referencing the same file. Many devices, such as a hard disk, may be referred to by more than one unit number, as they can hold many different files.

There are two I/O statements, `print` and a variant of `read`, that do not reference any unit number; these are the statements that we have used so far in examples, for the sake of simplicity. A `read` statement without a unit number will normally expect to read from the keyboard, unless the program is working in batch (non-interactive) mode, in which case there will be a disk file with a reserved name from which it reads. A `print` statement will normally expect to output to the screen, unless the program is in batch mode, in which case another disk file with a reserved name will be used. Such files are usually suitable for subsequent output on a physical output device. The system associates unit numbers with these default units (usually 5 for input and 6 for output), and their actual values may be accessed from the intrinsic module `iso_fortran_env`, see Section 9.24.

Apart from these two special cases, all I/O statements must refer explicitly to a unit in order to identify the device to which or from which data are to be transferred. The unit may be given in one of three forms. These are shown in the following examples, which use another form of `read` containing a unit specifier, *u*, and format specifier, *fmt*, in parentheses and separated by a comma, where *fmt* is a format specification as described in the previous section:

`read (`*u*`,` *fmt*`)` *list*

The three forms of *u* are as follows:

A scalar integer expression that gives the unit number:

```
read (4, '(f10.3)') q
read (nunit, '(f10.3)') q
read (4*i+j, 100) a
```

where the value may be any non-negative integer allowed by the system for this purpose.

An asterisk For example

```
read (*, '(f10.3)') q
```

where the asterisk implies the standard input unit designated by the system (`input_unit` in `iso_fortran_env`), the same as that used for `read` without a unit number.

A default character variable identifying an *internal file* (see next section).

A long-standing inconvenience in Fortran programs has been the need to manually manage input/output unit numbers. This becomes a real problem when using older third-party libraries that perform input/output and for which the source code is unavailable; when opening a file, it is not difficult to find a unit number that is not currently in use, but it may be the same as one that is employed later by other code. This can be overcome by the `newunit=` specifier on the `open` statement (Section 12.4). This returns a unique negative unit number on a successful `open`. Being negative, it cannot clash with any user-specified unit number (these being required to be non-negative), and the processor will choose a value that does not clash with anything it is using internally.

An example is:

```
integer :: unit
open (file='input.dat', newunit=unit)
⋮
read (unit, '(f10.3)') q
```

10.6 Internal files

Data can be converted to/from character form using a character variable as an **internal file**. Useful applications are to create a file name, create a character expression to use as a format, and prepare output lists containing mixed character and numerical data, all of which has to be prepared in character form, perhaps for output as a caption for a graph. The second application will now be described; the third will be dealt with in Section 10.8.

Imagine that we have to read a string of 30 digits, which might correspond to 30 one-digit integers, 15 two-digit integers, or ten three-digit integers. The information as to which type

of data is involved is given by the value of an additional digit, which has the value 1, 2, or 3, depending on the number of digits each integer contains. An internal file provides us with a mechanism whereby the 30 digits can be read into a character buffer area. The value of the final digit can be tested separately, and 30, 15, or ten values read from the internal file, depending on this value. The basic code to achieve this might read as follows (no error recovery or data validation is included, for simplicity):

```
integer      :: ival(30), key, i
character(30):: buffer
character(6) :: form(3) = (/ '(30i1)', '(15i2)', '(10i3)' /)
read (*, '(a30,i1)')      buffer, key
read (buffer, form (key)) (ival(i), i=1,30/key)
```

Here, `ival` is an array which will receive the values, `buffer` a character variable of a length sufficient to contain the 30 input digits, and `form` a character array containing the three possible formats to which the input data might correspond. The first `read` statement reads 30 digits into `buffer` as character data, and a final digit into the integer variable `key`. The second `read` statement reads the data from `buffer` into `ival`, using the appropriate conversion as specified by the edit descriptor selected by `key`. The number of variables read from `buffer` to `ival` is defined by the implied-`do` loop, whose second specifier is an integer expression depending also on `key`. After execution of this code, `ival` will contain `30/key` values, their number and exact format not having been known in advance.

If an internal file is a scalar, it has a single record whose length is that of the scalar. If it is an array, its elements, in array element order, are treated as successive records of the file and each has length equal to that of an array element. It may not be an array section with a vector subscript.

A record becomes defined when it is written. The number of characters sent must not exceed the length of the record. It may be less, in which case the rest of the record is padded with blanks. For list-directed output (Section 10.4), character constants are not delimited. A record may be read only if it is defined (which need not only be by an output statement).

An internal file is always positioned at the beginning of its first record prior to data transfer (the array section notation may be used to start elsewhere in an array). Of course, if an internal file is an allocatable array or pointer, it must be allocated or associated with a target. Also, no item in the input/output list may be in the file or associated with the file.

An internal file may be used for list-directed I/O (Section 10.9).

Also, ISO 10646 character variables (Section 3.7.2) may be used as internal files; and numeric, logical, default character, ASCII character, and ISO 10646 character values may all be read from or written to such a variable. For example,

```
subroutine japanese_date_stamp(string)
   integer, parameter :: ucs4 = selected_char_kind('ISO_10646')
   character(*, ucs4), intent(out) :: string
   integer                         :: val(8)
   call date_and_time(values=val)
   write (string, 10) val(1), '年', val(2), '月', val(3), '日'
10 format(i0,a,i0,a,i0,a)
end subroutine japanese_date_stamp
```

Note that, although reading from an ISO 10646 internal file into a default character or ASCII character variable is possible, it is only allowed when the data being read is representable in default character or ASCII character.

10.7 Formatted input

In the previous sections we have given complete descriptions of the ways that formats and units may be specified, using simplified forms of the `read` and `print` statements as examples. There are, in fact, two forms of the formatted `read` statement. Without a unit, it has the form

`read` *fmt [, input-list]*

and with a unit it has the form

`read (`*input-specifier-list*`)` *[input-list]*

where *input-list* is a list of variables and implied-`do` lists of variables. An *input-specifier* is one of:

[`unit=`*]u*	`decimal=`*expr*	`iostat=`*ios*
[`fmt=`*]fmt*	`end=`*end-label*	`pad=`*expr*
[`nml=`*]ngn*	`eor=`*eor-label*	`pos=`*expr*
`advance=`*expr*	`err=`*error-label*	`rec=`*index*
`asynchronous=`*expr*	`id=`*variable*	`round=`*expr*
`blank=`*expr*	`iomsg=`*message*	`size=`*size*

where *u* and *fmt* are the unit and format specifiers described in Sections 10.4 and 10.5, and *ngn* is a namelist group name. The unit specifier is required, and formatted I/O requires either a format specifier or a namelist group name; `nml=` is fully described in Section 10.10.

`iostat=`, `err=`, `end=`, and `iomsg=` are optional specifiers, described below in this section, that allow a user to specify how a `read` statement shall recover from various exceptional conditions. The optional specifiers `eor=` and `size=` are described in Section 10.11 and the optional specifier `rec=` is described in Section 10.13. The remaining optional specifiers are described in Chapters 11 and 12. The keyword items may be specified in any order, although it is usual to keep the unit number and format specification as the first two. The unit number must be first if it does not have its keyword. If the format does not have its keyword, it must be second, following the unit number without its keyword.

For simplicity of exposition, we have so far limited ourselves to formats that correspond to a single record in the file, but we will meet later in this chapter cases that lead to the input of part of a record or of several successive records.

The meanings of the optional specifiers are as follows:

- If `iostat=` appears, *ios* must be a scalar integer variable of default kind which, after execution of the `read` statement, has a negative value if an end-of-record condition is encountered during non-advancing input (Section 10.11), a different negative value if an endfile condition was detected on the input device (Section 12.2.3), a positive value if an error was detected (for instance a formatting error), or the value zero otherwise. The actual values assigned to *ios* in the event of an exception occurring are not defined by the standard, only the signs.

- If `end=` appears, *end-label* must be a statement label of a statement in the same scoping unit, to which control will be transferred in the event of the end of the file being reached.
- If `err=` appears, *error-label* is a statement label in the same scoping unit, to which control will be transferred in the event of any other exception occurring. The labels *error-label* and *end-label* may be the same. If they do not appear and an exception occurs, execution will stop, unless `iostat` is specified. An example of a `read` statement with its associated error recovery is given in Figure 10.3, in which `error` and `last_file` are subroutines to deal with the exceptions. They will normally be system dependent.
- If `iomsg=` appears, *message* identifies a scalar variable of type default character into which the processor places a message if an error, end-of-file, or end-of-record condition occurs during execution of the statement. If no such condition occurs, the value of the variable is not changed. Note that this is useful only for messages concerning error conditions, and an `iostat=` or `err=` specifier is needed to prevent an error causing immediate termination.

Figure 10.3 Testing for an error or the end of the file.

```
      read (nunit, '(3f10.3)', iostat=ios, err=110, end=120) a,b,c
   !  Successful read - continue execution.
      ⋮
 110  call error (ios) ! Error condition - take appropriate action.
      return
 120  call last_file   ! End of file  - test for more files.
      ⋮
```

If an error or end-of-file condition occurs on input, the statement terminates and all list items and any implied-`do` variables become undefined. If an end-of-file condition occurs for an external file, the file is positioned following the endfile record (Section 12.2.3); if there is otherwise an error condition, the file position is indeterminate. An end-of-file condition also occurs if an attempt is made to read beyond the end of an internal file.

It is a good practice to include some sort of error recovery in all `read` statements that are included permanently in a program. On the other hand, input for test purposes is normally sufficiently well handled by the simple form of `read` without a unit number, and without error recovery.

10.8 Formatted output

There are two types of formatted output statements, the `print` statement, which has appeared in many of the examples so far in this chapter, and the `write` statement, whose syntax is similar to that of the `read` statement:

`print` *fmt [, output-list]*

and

`write` (*output-specifier-list*) *[output-list]*

where *output-specifier* is one of:

[`unit=`*]u*	`decimal=`*expr*	`iostat=`*ios*
[`fmt=`*]fmt*	`delim=`*expr*	`pos=`*expr*
[`nml=`*]ngn*	`err=`*error-label*	`rec=`*index*
`advance=`*expr*	`id=`*variable*	`round=`*expr*
`asynchronous=`*expr*	`iomsg=`*message*	`sign=`*expr*

and the other syntax terms have the same meanings as described for the `read` statement (Section 10.7), or are described in Chapters 11 and 12. The optional specifier `rec=` is described in Section 10.13. Note that the unit specifier *u* is required, and the format specifier *fmt* is required unless for namelist I/O (Section 10.10). The optional `unit=` may be omitted only if the unit appears first, and the optional `fmt=` may be omitted only if it appears second and the optional `unit=` was also omitted. An asterisk for *u* specifies the standard output unit (`output_unit` in `iso_fortran_env`), as used by `print`. If an error condition occurs on output, execution of the statement terminates, any implied-`do` variables become undefined, and the file position becomes indeterminate.

An example of a `write` statement is

```
write (nout, '(10f10.3)', iostat=ios, err=110) a
```

An example using an internal file is given in Figure 10.4, which builds a character string from numeric and character components. The final character string might be passed to another subroutine for output, for instance as a caption for a graph.

Figure 10.4 Writing to an internal file.

```
integer           :: day
real              :: cash
character(len=50) :: line
⋮
!   write into line
write (line, '(a, i2, a, f8.2, a)')                    &
    'Takings for day ', day, ' are ', cash, ' dollars'
```

In this example we declare a character variable that is long enough to contain the text to be transferred to it. (The `write` statement contains a format specification with `a` edit descriptors without a field width. These assume a field width corresponding to the actual length of the character strings to be converted.) After execution of the `write` statement, `line` might contain the character string

```
Takings for day  3 are  4329.15 dollars
```

and this could be used as a string for further processing.

The number of characters written to `line` must not exceed its length.

10.9 List-directed I/O

In Section 10.4, the list-directed output facility using an asterisk as format specifier was introduced. We assumed that the list was short enough to fit into a single record, but for long lists the processor is free to output several records. Character constants may be split between records, and complex constants that are as long as, or longer than, a record may be split after the comma that separates the two parts. Apart from these cases, a value always lies within a single record. For historical reasons (carriage control, see Appendix B.5), the first character of each record is blank unless a delimited character constant is being continued. Note that when an undelimited character constant is continued, the first character of the continuation record is blank. The only blanks permitted in a numeric constant are within a split complex constant after the comma.

This facility is equally useful for input, especially of small quantities of test data. On the input record, the various constants may appear in most of their usual forms, just as if they were being read under the usual edit descriptors, as defined in Chapter 11. Exceptions are that complex values must consist of two numerical values separated by a comma and enclosed in parentheses, character constants may be delimited, a blank must not occur except in a delimited character constant or in a complex constant before or after a numeric field, blanks are never interpreted as zeros, and the optional characters that are allowed in a logical constant (those following `t` or `f`, see Section 11.3) must include neither a comma nor a slash. A complex constant spread over more than one record must have any end of record after the real part or before the imaginary part.

Character constants that are enclosed in apostrophes or quotation marks may be spread over as many records as necessary to contain them, except that a doubled quotation mark or apostrophe must not be split between records. Delimiters may be omitted for a default character constant if:

- it is of nonzero length;
- the constant does not contain a blank, comma, or slash;
- it is contained in one record;
- the first character is neither a quotation mark nor an apostrophe; and
- the leading characters are not numeric followed by an asterisk.

In this case, the constant is terminated when a blank, comma, slash, or end of record is encountered, and apostrophes or quotation marks appearing within the constant must not be doubled.

Whenever a character value has a different length from the corresponding list item, the value is truncated or padded on the right with blanks, as in the character assignment statement.

It is possible to use a repeat count for a given constant, for example `6*10` to specify six occurrences of the integer value 10. If it is possible to interpret the constant as either a literal constant or an undelimited character constant, the first corresponding list item determines which it is.

The (optionally repeated) constants are separated in the input by *separators*. A separator is one of the following, appearing other than in a character constant:

- a comma, optionally preceded and optionally followed by one or more contiguous blanks;
- a slash (/), optionally preceded and optionally followed by one or more contiguous blanks; or
- one or more contiguous blanks between two non-blank values or following the last non-blank value.

An end of record not within a character constant is regarded as a blank and, therefore, forms part of a separator. A blank embedded in a complex constant or delimited character constant is not a separator. An input record may be terminated by a slash separator, in which case all the following values in the record are ignored, and the input statement terminates.

If there are no values between two successive separators, or between the beginning of the first record and the first separator, this is taken to represent a **null value** and the corresponding item in the input list is left unchanged, defined or undefined as the case may be. A null value must not be used for the real or imaginary part of a complex constant, but a single null value may be used for the whole complex value. A series of null values may be represented by a repeat count without a constant: `,6*,`. When a slash separator is encountered, null values are given to any remaining list items.

An example of this form of the `read` statement is:

```
integer             :: i
real                :: a
complex             :: field(2)
logical             :: flag
character (len=12)  :: title
character (len=4)   :: word
⋮
read *, i, a, field, flag, title, word
```

If this reads the input record

`10`*b*`6.4`*b*`(1.,0.)`*b*`(2.,0.)`*b*`t`*b*`test/`

(in which *b* stands for a blank, and blanks are used as separators), `i`, `a`, `field`, `flag`, and `title` will acquire the values 10, 6.4, (1.,0.) and (2.,0.), `.true.`, and `test`, respectively, while `word` remains unchanged. For the input records

```
10,.64e1,2*,.true.
'histogramb10'/val1
```

(in which commas are used as separators), the variables `i`, `a`, `flag`, and `title` will acquire the values 10, 6.4, `.true.`, and `histogram`*b*10, respectively. The variables `field` and `word` remain unchanged, and the input string `val1` is ignored as it follows a slash. (Note the apostrophes, which are required as the string contains a blank. Without delimiters, this string would appear to be a string followed by the integer value 10.) Because of this slash, the read statement does not continue with the next record and the list is thus not fully satisfied.

10.10 Namelist I/O

It can be useful, especially for program testing, to input or output an annotated list of values to an external or internal file. The names of the objects involved are specified in a `namelist` group (Section 8.20), and the I/O is performed by a `read` or `write` statement that does not have an I/O list, and in which either

- the format is replaced by a namelist-group name as the second positional parameter; or
- the `fmt=` specifier is replaced by an `nml=` specifier with that name.

The sequence of records read or written by a namelist `read` or `write` statement begins with a record whose first non-blank character is an ampersand followed without an intervening blank by the group name, and ends with a record that terminates with a slash (`/`) that is not within a character constant. Each name or designator of an object is followed by an equals sign and a value or list of values, optionally preceded or followed by blanks. A value may be null. The objects with their values may appear in any order. The form of the list of values and null values is as that for list-directed I/O (Section 10.9), except that character constants must always be delimited in input records and logical constants must not contain an equals sign. A simple example is

```
integer   ::  no_of_eggs, litres_of_milk, kilos_of_butter
namelist/food/no_of_eggs, litres_of_milk, kilos_of_butter
read (5, nml=food)
```

to read the record

```
&food litres_of_milk=5, no_of_eggs=12 /
```

Note that the order of the two values given is not the same as their order in the `namelist` group. The value of `kilos_of_butter` remains unchanged.

On input, the designators of the objects are not case-sensitive and only those objects and subobjects that are specified in the input records and do not have a null value become defined. All other list items and parts of items remain in their existing state of definition or undefinition.

Where a subobject designator appears in an input record, all its substring expressions, subscripts, and strides must be scalar integer literal constants without specified kind parameters. All group names, object names, and component names are interpreted without regard to case. Blanks may precede or follow the name or designator, but must not appear within it.

If an object is scalar and of intrinsic type, the equals sign must be followed by one value. If it is an array, the equals sign must be followed by a list of values for its elements in array element order. If it is of derived type and defined derived-type I/O (Section 11.6) is not in use, the equals sign must be followed by a list of values of intrinsic type corresponding to its ultimate components in array element order. The list of values must not be too long, but it may be too short, in which case trailing null values are regarded as having been appended. If an object is of type character, the corresponding item must be of the same kind.

Zero-sized objects must not appear in a `namelist` input record. In any multiple occurrence of an object in a sequence of input records, the final value is taken.

Input records for `namelist` input may bear a comment following an object name/value separator other than a slash. As in the source form, it starts with an exclamation mark (`!`) and results in the rest of the record being ignored. This allows programmers to document the structure of a `namelist` input file line by line. For example, the input record of this section might be documented thus:

```
&food litres_of_milk=5,          ! For camping holiday
no_of_eggs=12 /
```

A comment line, with `!` as the first non-blank character in an input record, is also permitted, but in a character context such a line will be taken as part of the character value.

On output, all the objects in the `namelist` group are represented in the output records with names in upper case and are ordered as in the `namelist` group. Thus, the statements

```
integer   :: number, list(10)
namelist/out/number, list
write (6, nml=out)
```

might produce the record

```
&OUT NUMBER=1, LIST=14, 9*0 /
```

Repetitions in the `namelist` group result in a value for each occurrence.

10.11 Non-advancing I/O

So far we have considered each `read` or `write` statement to perform the input or output of a complete record. There are, however, many applications, especially in screen management, where this would become an irksome restriction. What is required is the ability to read and write without always advancing the file position to ahead of the next record. This facility is provided by **non-advancing I/O**. To gain access to this facility, the optional `advance=` specifier must appear in the `read` or `write` statement and be associated with a scalar default character expression *advance* which evaluates, after suppression of any trailing blanks and conversion of any upper-case letters to lower case, to the value `no`. The only other allowed value is `yes`, which is the default value if the specifier is absent; in this case, normal (advancing) I/O occurs.

The following optional specifiers are available for a non-advancing `read` statement:

`eor=`*eor-label*
`size=`*size*

where *eor-label* is a statement label in the same scoping unit and *size* is an integer scalar variable. The *eor-label* may be the same as an *end-label* or *error-label* of the `read` statement.

An advancing I/O statement always repositions the file after the last record accessed. A non-advancing I/O statement leaves the file positioned within the record, except that if it attempts to read data from beyond the end of the *current* record, an end-of-record condition occurs and the file is repositioned to follow the record. The `iostat` variable, if present, will acquire a negative value (`iostat_eor` in `iso_fortran_env`) that is different from the one indicating an end-of-file condition; and, if the `eor=` specifier is present, control is

transferred to the statement specified by its associated *eor-label*. In order to provide a means of controlling this process, the `size=` specifier, when present, sets *size* to the number of characters actually read. A full example is thus

```
      character(len=3) :: key
      integer          :: unit, size
      read (unit, '(a3)', advance='no', size=size, eor=66) key
      ⋮
  ! key is not in one record
66    key(size+1:) = ''
      ⋮
```

As for error and end-of-file conditions, the program terminates when an end-of-record condition occurs if neither `eor=` nor `iostat=` is specified.

If encountering an end-of-record on reading results in the input list not being satisfied, the `pad=` specifier described in Section 12.4 will determine whether any padding with blank characters occurs. Blanks inserted as padding are not included in the `size=` count.

It is possible to perform advancing and non-advancing I/O on the same record or file. For instance, a non-advancing read might read the first few characters of a record and an advancing read might read the remainder.

A particular application of this facility is to write a prompt to a screen and to read from the next character position on the screen without an intervening line-feed:

```
write (*, '(a)', advance='no') 'enter next prime number:'
read  (*, '(i10)') prime_number
```

Non-advancing I/O may be performed only on an external file, and may not be used for `namelist` or list-directed I/O. Note that, as for advancing input/output, several records may be processed by a single statement.

10.12 Unformatted I/O

This chapter has so far dealt with formatted I/O. The internal representation of a value may differ from the external form, which is always a character string contained in an input or output record. The use of formatted I/O involves an overhead for the conversion between the two forms, and often a round-off error too. There is also the disadvantage that the external representation usually occupies more space on a storage medium than the internal representation. These three drawbacks are all absent when unformatted I/O is used. In this form, the internal representation of a value is written exactly as it stands to the storage medium, and can be read back directly with neither round-off nor conversion overhead. Here, a value of derived type is treated as a whole and is not equivalent to a list of its ultimate components. This is another reason for the rule (Section 10.3) that it must not have an allocatable or pointer component at any level of component selection.

This type of I/O should be used in all cases where the records are generated by a program on one computer, to be read back on the same computer or another computer using the same internal number representations. Only when this is not the case, or when the data have to be

visualized in one form or another, should formatted I/O be used. The records of a file must all be formatted or all be unformatted (apart from the endfile record).

Unformatted I/O has the incidental advantage of being simpler to program since no complicated format specifications are required. The forms of the `read` and `write` statements are the same as for formatted I/O, but without any `fmt=` or `nml=` specifier:

```
read (4) q
write (nout, iostat=ios, err=110) a
```

The interpretation of `iostat=`, `err=`, and `end=` specifiers is as for formatted I/O.

Non-advancing I/O is not available (in fact, an `advance=` specifier is not allowed). Each `read` or `write` statement transfers exactly one record. The file must be an external file. On output to a file connected (Section 12.1) for sequential access, a record of sufficient length is created. On input, the type and type parameters of each entity in the list must agree with those of the value in the record, except that two reals may correspond to one complex when all three have the same kind parameter. The number of values specified by the input list of a `read` statement must not exceed the number of values available in the current record.

10.13 Direct-access files

The only type of file organization that we have so far dealt with is the sequential file which has a beginning and an end, and which contains a sequence of records, one after the other. Fortran permits another type of file organization known as **direct access** (or sometimes as random access or indexed). All the records have the same length, each record is identified by an index number, and it is possible to write, read, or rewrite any specified record without regard to position. (In a sequential file only the last record may be rewritten without losing other records; in general, records in sequential files cannot be replaced.) The records are either all formatted or all unformatted.

By default, any file used by a Fortran program is a sequential file. A direct-access file must be declared as such on its `open` statement (described Section 12.4) with the `access='direct'` and `recl=`*rl* specifiers (*rl* is the length of a record in the file). Once this declaration has been made, reading and writing, whether formatted or unformatted, proceeds as described for sequential files, except for the addition of a `rec=`*index* specifier to the `read` and `write` statements, where *index* is a scalar integer expression whose value is the index number of the record concerned. An `end=` specifier is not permitted. Usually, a data transfer statement for a direct-access file accesses a single record, but during formatted I/O any slash edit descriptor increases the record number by one and causes processing to continue at the beginning of this record. A sequence of statements to write, read, and replace a given record is given in Figure 10.5.

The file must be an external file and `namelist` formatting, list-directed formatting, and non-advancing I/O are all unavailable.

Direct-access files are particularly useful for applications that involve lots of hopping around inside a file, or where records need to be replaced, for instance in database applications. A weakness is that the length of all the records must be the same,[3] though,

[3] This deficiency is avoided with stream access, Section 10.16.

Figure 10.5 Write, read, and replace record 14. The `open` and `inquire` statements are explained in Sections 12.4 and 12.6.

```
integer, parameter :: nunit=2, len=100
integer            :: i, length
real               :: a(len), b(len+1:2*len)
   ⋮
inquire (iolength=length) a
open (nunit, access='direct', recl=length)
   ⋮
write (nunit, rec=14) b ! Write B to record 14 of direct-access file.
   ⋮
read (nunit, rec=14) a  ! Read the values back into array A.
   ⋮
do i = 1, len/2
   a(i) = i
end do
write (nunit, rec=14) a ! Replace record with new values.
```

on formatted output, the record is padded with blanks if necessary. For unformatted output, if the record is not filled, the remainder is undefined.

This simple and powerful facility allows much clearer control logic to be written than is the case for a sequential file which is repeatedly read, backspaced, or rewound. Only when sequential-access files become large may problems of long access times become evident on some computer systems, and this point should always be investigated before heavy investments are made in programming large direct-access file applications.

Some computer systems allow the same file to be regarded as sequential or direct access according to the specification in the `open` statement or its default. The standard, therefore, regards this as a property of the connection (Section 12.1) rather than of the file. In this case, the order of records, even for sequential I/O, is that determined by the direct-access record numbering.

10.14 UTF-8 files

The ISO 10646 standard specifies a standard encoding of UCS-4 characters into a stream of bytes, called UTF-8. Formatted files in UTF-8 format are supported by the `encoding=` specifier on the `open` statement (Section 12.4). For example,

```
open (20, name='output.file', action='write', encoding='utf-8')
```

The `encoding=` specifier on the `inquire` statement (Section 12.6) returns the encoding of a file, which will be `UTF-8` if the file is connected for UTF-8 input/output or the processor can detect the format in some way, `UNKNOWN` if the processor cannot detect the format, or

a processor-dependent value if the file is known to be in some other format (for example, `UTF-16LE`).

For the most part, UTF-8 files can be treated as ordinary formatted files. On output, all data is effectively converted to ISO 10646 characters for UTF-8 encoding.

On input, if data is being read into an ASCII character variable each input character must be in the range 0–127 (the ASCII subset of ISO 10646); if data is being read into a default character variable each input character must be representable in the default character set. These conditions will be satisfied if the data were written by numeric or logical formatting, or by character formatting from an ASCII or default character value; otherwise it would be safer to read the data into an ISO 10646 character variable for processing.

Figure 10.6 shows an example of the I/O procedures for a data processing application using these facilities.

Figure 10.6 Procedures for handling I/O for a data processing application.

```
subroutine write_id(unit, name, id)
  character(kind=ucs4, len=*), intent(in) :: name
  integer, intent(in)                     :: id, unit
  write (unit, '(1x,a,i6,2a)') 'Customer number ', id, ' is ', name
end subroutine write_id
⋮
subroutine read_id(unit, name, id)
  character(kind=ucs4, len=*), intent(out) :: name
  integer, intent(in)                      :: unit
  integer, intent(out)                     :: id
  character(kind=ucs4, len=20)             :: string
  integer                                  :: stringlen
  read (unit, '(1x,a16)', advance='no') string
  if (string/=ucs4_'Customer number ') stop 'Bad format'
  do stringlen=1, len(string)
    read (unit, '(3x,a)', advance='no') string(stringlen:stringlen)
    if (string(stringlen:stringlen)==ucs4_' ') exit
  end do
  read (string(1:stringlen), *) id
  read (unit, '(3x,a)') name
end subroutine read_id
```

Procedures for handling I/O for a data processing applicationŤ.

10.15 Asynchronous input/output

10.15.1 Asynchronous execution

Asynchronous input/output, long available as a compiler extension, has a standardized form, whereby other statements may execute while an input/output statement is in execution. It is

permitted only for external files opened with `asynchronous='yes'` in the `open` statement and is indicated by an `asynchronous='yes'` specifier in the `read` or `write` statement. By default, execution is synchronous even for a file opened with `asynchronous='yes'`, but it may be specified with `asynchronous='no'`. Execution of an asynchronous input/output statement initiates a 'pending' input/output operation and execution of other statements continues until it reaches a statement involving a wait operation for the file. This may be an explicit `wait` statement such as

```
wait (10)
```

or an `inquire` (Section 12.6), a `close` (Section 12.5), or a file positioning statement for the file. The compiler is permitted to treat each asynchronous input/output statement as an ordinary input/output statement (this, after all, is just the limiting case of the input/output being fast).

Here is a simple example:

```
real :: a(100000), b(100000)
open (10, file='mydata', asynchronous='yes')
read (10, '(10f8.3)', asynchronous='yes') a
   ⋮  ! Computation involving the array b
wait (10)
   ⋮  ! Computation involving the array a
```

Further asynchronous input/output statements may be executed for the file before the `wait` statement is reached. The input/output statements for each file are performed in the same order as they would have been if they were synchronous.

An execution of an asynchronous input/output statement may be identified by a scalar integer variable in an `id=` specifier. It must be of default kind or longer. Successful execution of the statement causes the variable to be given a processor-dependent value which can be passed to a subsequent `wait` or `inquire` statement as a scalar integer variable in an `id=` specifier.

A `wait` statement may have `end=`, `eor=`, `err=`, and `iostat=` specifiers. These have the same meanings as for a data transfer statement and refer to situations that occur while the input/output operation is pending. If there is also an `id=` specifier, only the identified pending operation is terminated and the other specifiers refer to this; otherwise, all pending operations for the file are terminated in turn.

An `inquire` statement is permitted to have a `pending=` specifier for a scalar default logical variable. If an `id=` specifier is present, the variable is given the value true if the particular input/output operation is still pending and false otherwise. If no `id=` specifier is present, the variable is given the value true if any input/output operations for the unit are still pending and false otherwise. In the 'false' case, wait operations are performed for the file or files. Wait operations are not performed in the 'true' case, even if some of the input/output operations are complete.

Execution of a `wait` statement specifying a unit that does not exist, has no file connected to it, or was not opened for asynchronous input/output is permitted, provided that the `wait` statement has no `id=` specifier; such a `wait` statement has no effect.

A file positioning statement (`backspace`, `endfile`, `rewind`, Section 12.2) performs wait operations for all pending input/output operations for the file.

Asynchronous input/output is not permitted in conjunction with defined derived-type I/O (Section 11.6) because it is anticipated that the number of characters actually written is likely to depend on the values of the variables.

A variable in a scoping unit is said to be an **affector** of a pending input/output operation if any part of it is associated with any part of an item in the input/output list, namelist, or `size=` specifier. While an input/output operation is pending, an affector is not permitted to be redefined, become undefined, or have its pointer association status changed. While an input operation is pending, an affector is also not permitted to be referenced or associated with a dummy argument with the `value` attribute (Section 19.8).

10.15.2 The asynchronous attribute

The `asynchronous` attribute for a variable warns the compiler that optimizations involving movement of code across `wait` statements (or other statements that cause wait operations) might lead to incorrect results. If a variable is a dummy argument or appears in an executable statement or a specification expression in a scoping unit and any statement of the scoping unit is executed while the variable is an affector, it must have the `asynchronous` attribute in the scoping unit.

A variable is automatically given this attribute if it or a subobject of it is an item in the input/output list, namelist, or `size=` specifier of an asynchronous input/output statement. A named variable may be declared with this attribute:

```
integer, asynchronous :: int_array(10)
```

or given it by the `asynchronous` statement

```
asynchronous :: int_array, another
```

This statement may be used to give the attribute to a variable that is accessed by use or host association.

Like the `volatile` attribute (Section 8.11.1), whether an object has the `asynchronous` attribute may vary between scoping units. If a variable is accessed by use or host association, it may gain the attribute, but it never loses it. For dummy and corresponding actual arguments, there is no requirement for agreement in respect of the `asynchronous` attribute. This provides useful flexibility, but needs to be used with care. If the programmer knows that all asynchronous action will be within the procedure, there is no need for the actual argument to have the `asynchronous` attribute. Similarly, if the programmer knows that no operation will ever be pending when the procedure is called, there is no need for the dummy argument to have the `asynchronous` attribute.

All subobjects of a variable with the `asynchronous` attribute have the attribute.

There are restrictions that avoid any copying of an actual argument when the corresponding dummy argument has the `asynchronous` attribute but does not have the `value` attribute: the actual argument must not be an array section with a vector subscript; if the actual argument is an array section or an assumed-shape array, the dummy argument must be an assumed-shape array; and if the actual argument is an array pointer, the dummy argument must be an array pointer or assumed-shape array.

10.16 Stream access files

Stream access is a further method of organizing an external file. It allows great flexibility for both formatted and unformatted input/output. It is established by specifying `access='stream'` on the `open` statement.

The file is positioned by **file storage units**, normally bytes, starting at position 1. The current position may be determined from a scalar integer variable in a `pos=` specifier of an `inquire` statement for the unit. A file may have the capability of positioning forwards or backwards, forwards only, or neither. If it has the capability, a required position may be indicated in a `read` or `write` statement by the `pos=` specifier, which accepts a scalar integer expression. In the absence of a `pos=` specifier, the file position is left unchanged.

It is the intention that unformatted stream input/output will read or write only the data to/from the file; that is, that there is no ancillary record length information (which is normally written for unformatted files). This allows easy interoperability with C binary streams, but the facility to skip or backspace over records is not available. If an output statement overwrites part of a file, the rest of the file is unchanged.

Here is a simple example of unformatted stream input/output:

```
real :: d
integer :: before_d
⋮
open (unit, ..., access='stream', form='unformatted')
⋮
inquire (unit, pos=before_d)
write (unit) d
⋮
write (unit, pos=before_d) d + 1
```

Assuming `d` occupies four bytes, the user could reasonably expect the first `write` to write exactly four bytes to the file. The use of the `pos=` specifier ensures that the second `write` will overwrite the previously written value of `d`.

Formatted **stream files** are very similar to ordinary (record-oriented) sequential files; the main difference is that there is no preset maximum record length (the `recl=` specifier in the `open` or `inquire` statements). If the file allows the relevant positioning, the value of a `pos=` specifier must be 1 or a value previously returned in an `inquire` statement for the file. As for a formatted sequential file, an output statement leaves the file ending with the data transferred.

Another difference from a formatted sequential file is that data-driven record termination in the style of C text streams is allowed. The intrinsic inquiry function `new_line(a)` (Section 9.8) returns the character that can be used to cause record termination. As an example, the following code will write two lines to the file `/dev/tty`:

```
open (28, file='/dev/tty', access='stream', form='formatted')
write (28, '(a)') 'Hello'//new_line('x')//'World'
```

10.17 Execution of a data transfer statement

So far, we have used simple illustrations of data transfer statements without dependencies. However, some forms of dependency are permitted and can be very useful. For example, the statement

```
read (*, *) n, a(1:n)                ! n is an integer
```

allows the length of an array section to be part of the data.

With dependencies in mind, the order in which operations are executed is important. It is as follows:

i) identify the unit;

ii) establish the format (if any);

iii) position the file ready for the transfer (if required);

iv) transfer data between the file and the I/O list or namelist;

v) position the file following the transfer (if required);

vi) cause the `iostat` and `size` variables (if present) to become defined.

The order of transfer of namelist input is that in the input records. Otherwise, the order is that of the I/O list or `namelist`. Each input item is processed in turn, and may affect later subobjects and implied-`do` indices. All expressions within an I/O list item are determined at the beginning of the processing of the item. If an entity is specified more than once during execution of a namelist input statement, the later value overwrites the earlier value. Any zero-sized array or zero-length implied-`do` list is ignored.

When an input item is an array, no element of the array is permitted to affect the value of an expression within the item. For example, the cases shown in Figure 10.7 are not permitted. This prevents dependencies occurring within the item itself.

Figure 10.7 Dependencies are not permitted within an input item.

```
integer :: j(10)
⋮
read *, j(j)                          ! Not permitted
read *, j(j(1):j(10))                 ! Not permitted
```

In the case of an internal file, an I/O item must not be in the file or associated with it. Nor may an input item contain or be associated with any portion of the established format.

Recursive I/O, that is, a function referenced in an I/O statement executing an I/O statement for the same unit, is discussed in Section 11.7.

10.18 Summary

This chapter has begun the description of Fortran's extensive I/O facilities. It has covered the formatted I/O statements, and then turned to unformatted I/O and direct-access, UTF-8, and asynchronous files, as well as stream I/O.

The syntax of the `read` and `write` statements has been introduced gradually. The full syntax, excluding `read` without an explicit unit, is

`read (`*input-specifier-list*`)` *[input-list]*

and

`write (`*output-specifier-list*`)` *[output-list]*

An input or output specifier list must include a unit specifier and must not include any specifier more than once. Specifier variables, as in `iostat=` and `size=`, must not be associated with each other (for instance, be identical), nor with any entity being transferred, nor with any *do-var* of an implied-`do` list of the same statement. If either of these variables is an array element, the subscript value must not be affected by the data transfer, implied-`do` processing, or the evaluation of any other specifier in the statement.

Exercises

1. Write statements to output the state of a game of tic-tac-toe (noughts and crosses) to a unit designated by the variable `unit`.
2. Write a program which reads an input record of up to 132 characters into an internal file and classifies it as a Fortran comment line with no statement, an initial line without a statement label, an initial line with a statement label, a continuation line, or a line containing multiple statements.
3. Write a subroutine `get_char(unit,c,end_of_file)` to read a single character `c` from a formatted, sequential file `unit`, ignoring any record structure; `end_of_file` is a logical variable that is given the value `.true.` if the end of the file is reached and the value `.false.` otherwise.
4. Write an input procedure that reads a variable number of characters from a file, stopping on encountering a character in a user-specified set or at end of record, returning the input in a deferred-length allocatable character string.
5. Write a program that reads a file (presumed to be a text file) as an unformatted stream, checking for Unix (LF) and DOS/Windows (CRLF) record terminators.

11. Edit descriptors

11.1 Introduction

In the description of number conversion in Section 10.2, a few examples of the edit descriptors were given. As mentioned there, edit descriptors give a precise specification of how values are to be converted into a character string on an output device or internal file, or converted from a character string on an input device or internal file to internal representations.

With certain exceptions noted in the following text, edit descriptors in a list are separated by commas.

Edit descriptors are interpreted without regard to case. This is also true for numerical and logical input fields; an example is `89AB` as a hexadecimal input value. In output fields, any alphabetic characters are in upper case.

Edit descriptors fall into three classes: **character string**, **data**, and **control**.

11.2 Character string edit descriptor

A character literal constant without a specified kind parameter (thus of default kind) can be transferred to an output file by embedding it in the format specification itself, as in the example

```
print "(' This is a format statement')"
```

The string will appear each time it is encountered during format processing. In this descriptor, case is significant. Character string edit descriptors must not be used on input.

11.3 Data edit descriptors

Data edit descriptors are edit descriptors that transfer data between the input/output list and an internal or external file. A format specification may contain no data edit descriptor if there is no data in the input/output list, for example, if the input/output list is empty or consists entirely of zero-sized arrays.

No form of value of any of the intrinsic data types, on either input or output, may be with a kind type parameter. For all the numeric edit descriptors, if an output field is too narrow to contain the number to be output, it is filled with asterisks.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

11.3.1 Repeat counts

A data edit descriptor may be preceded by a **repeat count** (a nonzero unsigned default integer literal constant), as in the example

```
10f12.3
```

A repeat count may also be applied directly to the slash edit descriptor (Section 11.4.3), but not to any other control edit descriptor or to a character string edit descriptor. However, a repeat count may be applied to a **subformat**, that is, a group of edit descriptors enclosed in parentheses:

```
print '(4(i5,f8.2))', (i(j), a(j), j=1,4)
```

(for integer `i` and real `a`). This is equivalent to writing

```
print '(i5,f8.2,i5,f8.2,i5,f8.2,i5,f8.2)', (i(j), a(j), j=1,4)
```

Repeat counts such as this may be nested:

```
print '(2(2i5,2f8.2))', i(1),i(2),a(1),a(2),i(3),i(4),a(3),a(4)
```

If a format specification without a subformat in parentheses is used with an I/O list that contains more elements than the number of edit descriptors, taking account of repeat counts, then a new record will begin, and the format specification will be repeated. Further records begin in the same way until the list is exhausted. To print an array of 100 integer elements, 10 elements to a line, the following statement might be used:

```
print '(10i8)', i(1:100)
```

Similarly, when reading from an input file, new records will be read until the list is satisfied, a new record being taken from the input file each time the specification is repeated *even if the individual records contain more input data than specified by the format specification*. These superfluous data will be ignored. For example, reading the two records (*b* again stands for a blank)

*bbb*10*bbb*15*bbb*20
*bbb*25*bbb*30*bbb*35

under control of the `read` statement

```
read '(2i5)', i,j,k,l
```

will result in the four integer variables `i`, `j`, `k`, and `l` acquiring the values 10, 15, 25, and 30, respectively.

If a format contains a subformat in parentheses, as in

```
'(2i5, 3(i2,2(i1,i3)), 2(2f8.2,i2))'
```

whenever the format is exhausted, a new record is taken and format control reverts to the repeat factor preceding the left parenthesis corresponding to the last-but-one right parenthesis, here `2(2f8.2,i2)`, or to the left parenthesis itself if it has no repeat factor. This we call **reversion**.

In order to make the writing of CSV (comma-separated values) files easier, **unlimited format repetition** can be used to repeat a format specification without any preset limit, as

long as there are still data left to transfer to or from the I/O list. This is specified by `*` (*format-items*), and is permitted only as the last item in a format specification. A colon edit descriptor (Section 11.4.4) may be needed to avoid unwanted data from character string edit descriptors after the final data item. Here is an example:

```
print '("List: ",*(g0,:,", "))', 10, 20, 30
```

will print

```
 List: 10, 20, 30
```

(the `g0` edit decriptor is described in Section 11.3.7).

11.3.2 Integer formatting

Integer values may be converted by means of the `i` edit descriptor. Its basic form is `i`*w*, where *w* is a nonzero unsigned default integer literal constant that defines the width of the field. The integer value will be read from or written to this field, adjusted to its right-hand side. If we again designate a blank position by *b* then the value −99 printed under control of the edit descriptor `i5` will appear as *bb*`-99`, the sign counting as one position in the field.

For output, an alternative form of this edit descriptor allows the number of digits that are to be printed to be specified exactly, even if some are leading zeros. The form `i`*w.m* specifies the width of the field, *w*, and that at least *m* digits are to be output, where *m* is an unsigned default integer literal constant. The value 99 printed under control of the edit descriptor `i5.3` would appear as *bb*`099`. The value of *m* is even permitted to be zero, and the field will be then filled with blanks if the value printed is 0. On input, `i`*w.m* is interpreted in exactly the same way as `i`*w*.

In order to allow output records to contain as little unused space as possible, the `i` edit descriptor may specify *w* to be zero, as in `i0`. This does not denote a zero-width field, but a field that is of the minimum width necessary to contain the output value in question. The programmer does not need to worry that a field with too narrow a width will cause an output field to overflow and contain only asterisks.

Integer values may also be converted by the `b`*w*, `b`*w.m*, `o`*w*, `o`*w.m*, `z`*w*, and `z`*w.m* edit descriptors. These are similar to the `i` form, but are intended for integers represented in the binary, octal, and hexadecimal number systems, respectively (Section 2.6.6). The external form does not contain the leading letter (`b`, `o`, or `z`) or the delimiters. The *w.m* form, with *m* equal to *w*, is recommended on output, so that any leading zeros are visible.

11.3.3 Real formatting

Real values may be converted by the `e`, `en`, `es`, or `f` edit descriptors.[1] The `f` descriptor we have met in earlier examples. Its general form is `f`*w.d*, where *w* and *d* are unsigned default integer literal constants which define, respectively, the field width and the number of digits to appear after the decimal point in the output field. For input, *w* must not be zero. The decimal point counts as one position in the field. On input, if the input string has a decimal point, the value of *d* is ignored. Reading the input string *b*`9.3729`*b* with the edit descriptor `f8.3`

[1] And in Fortran 2018 by the `ex` edit descriptor, see Section 22.13.

would cause the value 9.3729 to be transferred. All the digits are used, but round-off may be inevitable because of the actual physical storage reserved for the value on the computer being used.

There are, in addition, two other forms of input string that are acceptable to the `f` edit descriptor. The first is an optionally signed string of digits without a decimal point. In this case, the *d* rightmost digits will be taken to be the fractional part of the value. Thus, *b*`-14629` read under control of the edit descriptor `f7.2` will transfer the value −146.29. The second form is the standard default real form of a literal constant, as defined in Section 2.6.2, and the variant in which the exponent is signed and `e` is omitted. In this case, the *d* part of the descriptor is again ignored. Thus, the value `14.629e-2` (or `14.629-2`), under control of the edit descriptor `f9.1`, will transfer the value 0.14629. The exponent letter may also be written in upper case.

Values are rounded on output following the normal rules of arithmetic. Thus, the value 10.9336, when output under control of the edit descriptor `f8.3`, will appear as *bb*`10.934`, and under the control of `f4.0` as *b*11. For output, if *w* is zero, as in `f0.3`, this denotes a field that is of the minimum width necessary to contain the output value in question.

The `e` edit descriptor has two forms, e*w.d* and e*w.d*e*e*, and is more appropriate for numbers with a magnitude below about 0.01, or above 1000. The value of *w* must not be zero. The rules for these two forms for input are identical to those for the f*w.d* edit descriptor. For output with the e*w.d* form of the descriptor, a different character string will be transferred, containing a significand with absolute value less than 1 and an exponent field of four characters that consists of either `E` followed by a sign and two digits or of a sign and three digits. Thus, for 1.234×10^{23} converted by the edit descriptor `e10.4`, the string *b*`.1234E+24` or *b*`.1234+024` will be transferred. The form containing the exponent letter `E` is not used if the magnitude of the exponent exceeds 99. For instance, `e10.4` would cause the value 1.234×10^{-150} to be transferred as *b*`.1234-149`. Some processors print a zero before the decimal point.

In the second form of the `e` edit descriptor, e*w.d*e*e*, *e* is an unsigned, nonzero default integer literal constant that determines the number of digits to appear in the exponent field. This form is obligatory for exponents whose magnitude is greater than 999. Thus, the value 1.234×10^{1234} with the edit descriptor `e12.4e4` is transferred as the string *b*`.1234E+1235`. An increasing number of computers are able to deal with these very large exponent ranges. It can also be used if only one exponent digit is desired. For example, the value 1.211 with the edit descriptor `e9.3e1` is transferred as the string *b*`0.121E+1`.

The `en` (**engineering**) edit descriptor is identical to the `e` edit descriptor except that on output the decimal exponent is divisible by three, a nonzero significand is greater than or equal to 1 and less than 1000, and the scale factor (Section 11.4.1) has no effect. Thus, the value 0.0217 transferred under an `en9.2` edit descriptor would appear as `21.70E-03` or `21.70-003`.

The `es` (**scientific**) edit descriptor is identical to the `e` edit descriptor, except that on output the absolute value of a nonzero significand is greater than or equal to 1 and less than 10, and the scale factor (Section 11.4.1) has no effect. Thus, the value 0.0217 transferred under an `es9.2` edit descriptor would appear as `2.17E-02` or `2.17-002`.

On a computer with IEEE arithmetic, all the edit descriptors for reals treat IEEE exceptional values in the same way and only the field width `w` is taken into account.

The output forms, each right justified in its field, are

i) `-Inf` or `-Infinity` for minus infinity;

ii) `Inf`, `+Inf`, `Infinity`, or `+Infinity` for plus infinity; and

iii) `NaN`, optionally followed by alphanumeric characters in parentheses (to hold additional information).

On input, upper- and lower-case letters are treated as equivalent. The forms are

i) `-Inf` or `-Infinity` for minus infinity;

ii) `Inf`, `+Inf`, `Infinity`, or `+Infinity` for plus infinity; and

iii) `NaN`, optionally followed by alphanumeric characters in parentheses for a NaN. With no such alphanumeric characters it is a quiet NaN.

11.3.4 Complex formatting

Complex values may be edited under control of pairs of `f`, `e`, `en`, or `es` edit descriptors. The two descriptors do not need to be identical. The complex value (0.1,100.) converted under control of `f6.1,e8.1` would appear as *bbb*`0.1`*b*`0.1E+03`. The two descriptors may be separated by character string and control edit descriptors (to be described in Sections 11.2 and 11.4, respectively).

11.3.5 Logical formatting

Logical values may be edited using the `l`*w* edit descriptor. This defines a field of width *w* which on input consists of optional blanks, optionally followed by a decimal point, followed by `t` or `f` (or `T` or `F`), optionally followed by additional characters. Thus, a field defined by `l7` permits the strings `.true.` and `.false.` to be input. The characters `t` or `f` will be transferred as the values true or false, respectively. On output, the character `T` or `F` will appear in the rightmost position in the output field.

11.3.6 Character formatting

Character values may be edited using the `a` edit descriptor in one of its two forms, either `a` or `a`*w*. In the first of the two forms, the width of the input or output field is determined by the actual width of the item in the I/O list, measured in number of characters of whatever kind. Thus, a character variable of length 10, containing the value STATEMENTS, when written under control of the `a` edit descriptor would appear in a field ten characters wide, and the non-default character variable of length 4 containing the value 国際標準 would appear in a field four characters wide. If, however, the first variable were converted under an `a11` edit descriptor, it would be printed with a leading blank: *b*`STATEMENTS`. Under control of `a8`, the eight leftmost characters only would be written: `STATEMEN`.

Conversely, with the same variable on input, an `a11` edit descriptor would cause the ten rightmost characters in the 11-character-wide input field to be transferred, so that *b*`STATEMENTS` would be transferred as `STATEMENTS`. The `a8` edit descriptor would cause the eight characters in the field to be transferred to the eight leftmost positions in the variable, and the remaining two would be filled with blanks: STATEMEN would be transferred as `STATEMEN`*bb*.

Whenever an input list and the associated format specify more data than appear in the record, any padding will be controlled by the `pad=` specifier currently in effect (Section 12.4).

All characters transferred under the control of an `a` or a*w* edit descriptor have the kind of the I/O list item, and we note that this edit descriptor is the *only* one which can be used to transmit non-default characters to or from a record. In the non-default case, the blank padding character is processor dependent.

11.3.7 General formatting

Any intrinsic data type values may be edited with the g*w.d* or g*w.d*e*e* (**general**) edit descriptor. When used for real or complex types it is identical to the e*w.d* or e*w.d*e*e* edit descriptor, except for output when `f` format is suitable for representing exactly the decimal value N obtained by rounding the internal value to d significant decimal digits according to the I/O rounding mode in effect. For example, the edit descriptor `g10.3` outputs the values 0.01732, 1.732, 173.2, and 1732.0 as `0.173e-01`, `1.73`, `173.`, and `0.173e04`, respectively. The formal rule for which form is employed is as follows. Let s be 1 if N is zero and otherwise be the integer such that

$$10^{s-1} \leq N < 10^{s}.$$

If $0 \leq s \leq d$, then `f`$(w-n).(d-s)$ formatting followed by n blanks is used, where n=4 for g*w.d* editing and e+2 for g*w.d*e*e* editing.

This form is useful for printing values whose magnitudes are not well known in advance, where `f` conversion is preferred when possible and `e` conversion otherwise.

When the `g` edit descriptor is used for integer, logical, or character types, it follows the rules of the i*w*, `l`*w*, and a*w* edit descriptors, respectively (any d or e is ignored).

In order to make the writing of CSV (comma-separated values) files easier, the `g0` edit descriptor may be used; this transfers the user data as follows:

- integer data are transferred as if `i0` had been specified;
- real and complex data are transferred as if es*w*`.`*d*e*e* had been specified, where the compiler chooses the values of w, d, and e depending on the actual value to be output;
- logical data are transferred as if `l1` had been specified;
- character data are transferred as if `a` had been specified.

For example,

```
print '(1x, 5(g0, "; "))', 17, 2.71828, .false., "Hello"
```

will print something like

```
17; 2.7183e+00; F; Hello;
```

(depending on the values for *w*, *d*, and *e* chosen by the compiler for the floating-point datum).

The `g0.d` edit descriptor is similar to the `g0` edit descriptor but specifies the value to be used as *d* for floating-point data; as seen in the example above, this can be necessary if the value chosen by the compiler is unsuitable. Unfortunately, the standard forbids the use of `g0.d` for anything other than floating-point data, even though *d* is ignored for `gw.d` for those data types, removing much of its convenience at a stroke. [2]

11.3.8 Derived type formatting

Derived-type values may be edited by the appropriate sequence of edit descriptors corresponding to the intrinsic types of the ultimate components of the derived type. An example is:

```
type string
   integer           :: length
   character(len=20) :: word
end type string
type(string) :: text
read (*, '(i2, a)') text
```

They may also be edited by edit descriptors written by the programmer (see Section 11.6).

11.4 Control edit descriptors

It is sometimes necessary to give other instructions to an I/O device than just the width of fields and how the contents of these fields are to be interpreted. For instance, it may be that one wishes to position fields at certain columns or to start a new record without issuing a new `write` command. For this type of purpose, the control edit descriptors provide a means of telling the processor which action to take. Some of these edit descriptors contain information that is used as it is processed; others are like switches, which change the conditions under which I/O takes place from the point where they are encountered until the end of the processing of the I/O statement containing them (including reversions, Section 11.3.1). The latter descriptors are further divided into ones which alter the changeable connection modes (described in Section 11.5), and the irregular descriptor for changing the scale factor, which we will deal with first.

11.4.1 Scale factor

The scale factor applies to the input of real quantities under the `e`, `f`, `en`, `es`, and `g` edit descriptors,[3] and are a means of scaling the input values. Their form is *k*`p`, where *k* is a default integer literal constant. The scale factor is zero at the beginning of execution of the statement. The effect is that any quantity which does not have an exponent field will be reduced by a factor 10^k. Quantities with an exponent are not affected.

[2] This has been corrected in Fortran 2018, see Section 23.15.1.

[3] And in Fortran 2018, the `ex` edit descriptor.

The scale factor *k*p also affects output with e, f, or g editing, but has no effect with en or es editing.[4] Under control of an f edit descriptor, the quantity will be multiplied by a factor 10^k. Thus, the number 10.39 output by an f6.0 edit descriptor following the scale factor 2p will appear as *b*1039.. With the e edit descriptor, and with g where the e style editing is taken, the quantity is transferred as follows: if $0 < k < d+2$, the output field contains exactly k significant digits to the left of the decimal point and $d-k+1$ significant digits to its right; if $-d < k \leq 0$, the output field has a zero to the left of the decimal point, and to its right has $|k|$ zeros followed by $d-|k|$ significant digits. Other values of k are not permitted. Thus 310.0, written with 2p,e9.2 editing, will appear as 31.0E+01. This gives better control over the output style of real quantities which otherwise would have no significant digits before the decimal point.

The comma between a scale factor and an immediately following f, e, en, es, or g edit descriptor (with or without a repeat count) may be omitted, but we do not recommend this since it suggests that the scale factor applies only to the next edit descriptor, whereas in fact it applies throughout the format until another scale factor is encountered.

11.4.2 Tabulation and spacing

Tabulation in an input or output field can be achieved using the edit descriptors t*n*, tr*n*, and tl*n*, where *n* is a positive default integer literal constant. These state, respectively, that the next part of the I/O should begin at position *n* in the current record (where the **left tab limit** is position 1), or at *n* positions to the right of the current position, or at *n* positions to the left of the current position (the left tab limit if the current position is less than or equal to *n*). Let us suppose that, following an advancing read, we read an input record *bb*9876 with the following statement:

```
read (*, '(t3, i4, tl4, i1, i2)') i, j, k
```

The format specification will move a notional pointer firstly to position 3, whence i will be read. The variable i will acquire the value 9876, and the notional pointer is then at position 7. The edit descriptor tl4 moves it left four positions, back to position 3. The quantities j and k are then read, and they acquire the values 9 and 87, respectively. These edit descriptors cause replacement on output, or multiple reading of the same items in a record on input. On output, any gaps ahead of the last character actually written are filled with spaces. If any character that is skipped by one of the descriptors is of other than default type, the positioning is processor dependent.

If the current record is the first one processed by the I/O statement and follows non-advancing I/O that left the file positioned within a record, the next character is the left tab limit; otherwise, the first character of the record is the left tab limit.

The *n*x edit descriptor is equivalent to the tr*n* edit descriptor. It is often used to place spaces in an output record. For example, to start an output record with a blank by this method, one writes

```
fmt= '(1x,....)'
```

[4] In Fortran 2018 it also has no effect on output with ex editing.

Spaces such as this can precede a data edit descriptor, but `1x,i5` is not, for instance, exactly equivalent to `i6` on output, as any value requiring the full six positions in the field will not have them available in the former case.

The `t` and `x` edit descriptors never cause replacement of a character already in an output record, but merely cause a change in the position within the record such that such a replacement might be caused by a subsequent edit descriptor.

11.4.3 New records (slash editing)

New records may be started at any point in a format specification by means of the slash (`/`) edit descriptor. This edit descriptor, although described here, may in fact have repeat counts; to skip, say, three records one can write either `/,/,/` or `3/`. On input, a new record will be started each time a `/` is encountered, even if the contents of the current record have not all been transferred. Reading the two records

bbb`99`*bbb*`10`
bb`100`*bbb*`11`

with the statement

```
read '(bz,i5,i3,/,i5,i3,i2)', i, j, k, l, m
```

will cause the values 99, 0, 100, 0, and 11 to be transferred to the five integer variables, respectively. This edit descriptor does not need to be separated by a comma from a preceding edit descriptor, unless it has a repeat count; it does not ever need to be separated by a comma from a succeeding edit descriptor.

The result of writing with a format containing a sequence of, say, four slashes, as represented by

```
print '(i5,4/,i5)', i, j
```

is to separate the two values by three blank records (the last slash starts the record containing `j`); if `i` and `j` have the values 99 and 100, they would appear as

bbb`99`
b
b
b
bb`100`

A slash edit descriptor written to an internal file will cause the following values to be written to the next element of the character array specified for the file. Each such element corresponds to a record, and the number of characters written to a record must not exceed its length.

11.4.4 Colon editing

Colon editing is a means of terminating format control if there are no further items in an I/O list. In particular, it is useful for preventing further output of character strings used for annotation if the output list is exhausted. Consider the following output statement, for an array `l(3)`:

```
print '(" l1 = ", i5, :, " l2 = ", i5, :," l3 = ", i5)', &
       (l(i) ,i=1,n)
```

If `n` has the value 3, then three values are printed. If `n` has the value 1, then, without the colons, the following output string would be printed:

```
l1 = 59 l2 =
```

The colon, however, stops the processing of the format, so that the annotation for the absent second value is not printed. This edit descriptor need not be separated from a neighbour by a comma. It has no effect if there are further items in the I/O list.

11.5 Changeable file connection modes

A file connected for formatted input/output has several modes that affect the processing of data edit descriptors. These modes can be specified (or changed) by an `open` statement (Section 12.4), overridden for the duration of a data transfer statement execution by a `read` or `write` statement (Sections 10.7 and 10.8), or changed within a format specification by an edit descriptor.

Note that, as usual, the specifiers on an `open`, `read`, or `write` statement take default character values, and are not case-sensitive.

For an internal file there is of course no `open` statement, so these have a predetermined value at the beginning of every `read` or `write` statement.

11.5.1 Embedded blank interpretation

Embedded blanks in numeric input fields are treated in one of two ways, either as zero, or as null characters that are squeezed out by moving the other characters in the input field to the right, and adding leading blanks to the field (unless the field is totally blank, in which case it is interpreted as zero). This is controlled by the **blank interpretation mode**.

The blank interpretation mode for a unit may be controlled by the `blank=` specifier on the `open` statement, which takes one of the values `null` or `zero`. If not specified, or for an internal file, the default is `null`. It may be overridden by a `blank=` specifier in a `read` statement.

There is a corresponding specifier in an `inquire` statement (Section 12.6) that is assigned the value `NULL`, `ZERO`, or `UNDEFINED` as appropriate.

The blank interpretation mode may be temporarily changed within a `read` statement by the `bn` (blanks null) and `bz` (blanks zero) edit descriptors.

Let us suppose that the mode is that blanks are treated as zeros. The input string *bb*1*b*4 converted by the edit descriptor `i5` would transfer the value 104. The same string converted by `bn,i5` would give 14. A `bn` or `bz` edit descriptor switches the mode for the rest of that format specification, or until another `bn` or `bz` edit descriptor is met. The `bn` and `bz` edit descriptors have no effect on output.

11.5.2 Input/output rounding mode

Rounding during formatted input/output is controlled by the **input/output rounding mode**, which is one of `up`, `down`, `zero`, `nearest`, `compatible`, or `processor_defined`. The meanings are obvious except for the difference between `nearest` and `compatible`. Both refer to a closest representable value, but if two are equidistant, which is taken is processor dependent for `nearest`[5] and the value away from zero for `compatible`. An input/output rounding mode of `processor_defined` places no constraints on what kind of rounding will be performed by the processor.

The input/output rounding mode may be controlled by the `round=` specifier on the `open` statement, which takes one of the values `up`, `down`, `zero`, `nearest`, `compatible`, or `processor_defined`. If it is not specified, or for an internal file, the mode will be one of the above, but which one is processor dependent. It may be overridden by a `round=` specifier in a `read` or `write` statement with one of these values.

There is a corresponding specifier in the `inquire` statement that is assigned the value `UP`, `DOWN`, `ZERO`, `NEAREST`, `COMPATIBLE`, `PROCESSOR_DEFINED`, or `UNDEFINED`, as appropriate. The processor returns the value `PROCESSOR_DEFINED` only if the input/output rounding mode currently in effect behaves differently from the other rounding modes.

The input/output rounding mode may be temporarily changed within a `read` or `write` statement to `up`, `down`, `zero`, `nearest`, `compatible`, or `processor_defined` by the `ru`, `rd`, `rz`, `rn`, `rc`, or `rp` edit descriptor, respectively.

11.5.3 Signs on positive values

Leading signs are always written for negative numerical values on output. For positive quantities other than exponents, the plus signs are optional, and whether the signs are written depends on the **sign mode**. If the mode is `suppress`, leading plus signs are suppressed, that is the value 99 printed by `i5` is *bbb*`99` and 1.4 is printed by `e10.2` as *bb*`0.14E+01`. If the mode is `plus`, leading plus signs are printed; the same numbers written in this mode become *bb*`+99` and *b*`+0.14E+01`. If the mode is `processor_defined` (the default), it is processor dependent whether leading plus signs are printed or suppressed.

The sign mode may be specified by the `sign=` specifier on an `open` statement, which can take the value `suppress`, `plus`, or `processor_defined`. If no `sign=` specifier appears, or for output to an internal file, the sign mode is `processor_defined`.

The sign mode for a connection may be overridden by a `sign=` specifier in a `write` statement with one of these values.

There is a corresponding specifier in the `inquire` statement that is assigned the value `PLUS`, `SUPPRESS`, `PROCESSOR_DEFINED`, or `UNDEFINED`, as appropriate.

The sign mode may also be temporarily changed within a `write` statement by the `ss`, `sp`, and `s` edit descriptors. The `ss` (sign suppress) edit descriptor changes the mode to `suppress`, suppressing leading plus signs. To switch on plus sign printing, the `sp` (sign print) edit descriptor may be used. Thus, printing the values 99 and 1.4 with the edit descriptors `ss,i5,sp,e10.2` will print *bbb*`99` and *b*`+0.14E+01` regardless of the current sign mode. An

[5] If the processor supports IEEE arithmetic I/O conversions, `nearest` should be round to even, the same as IEEE arithmetic rounding.

`ss`, `sp`, or `s` will remain in force for the remainder of the format specification, unless another `ss`, `sp`, or `s` edit descriptor is met.

The `sign=` specifier and these edit descriptors provide complete control over sign printing, and are useful for producing coded outputs which have to be compared automatically on two different computers.

11.5.4 Decimal comma for input/output

Many countries use a decimal comma instead of a decimal point. Support for this is provided by the `decimal=` input/output specifier and by the `dc` and `dp` edit descriptors. These affect the **decimal edit mode** for the unit. While the decimal edit mode is `point`, decimal points are used in input/output. This is the default mode.

While the mode is `comma`, commas are used in place of decimal points both for input and for output. For example,

```
x = 22./7
print '(1x,f6.2)', x
```

would produce the output

```
3,14
```

in `comma` mode.

The `decimal=` specifier may appear on the `open` (Section 12.4), `read`, and `write` statements, and has the form

`decimal=`*scalar-character-expr*

where the *scalar-character-expr* evaluates either to `point` or to `comma`. On the `open` statement it specifies the default decimal edit mode for the unit. With no `decimal=` clause on the `open` statement, or for an internal file, the default is `point`. On an individual `read` or `write` statement, the `decimal=` clause specifies the default mode for the duration of that input/output statement only.

The `dc` and `dp` edit descriptors change the decimal edit mode to `comma` and `point`, respectively. They take effect when they are encountered during format processing and continue in effect until another `dc` or `dp` edit descriptor is encountered or until the end of the current input/output statement. For example,

```
   write (*,10) x, x, x
10 format(1x,'Default ',f5.2,', English ',dp,f5.2,'Français',dc,f5.2)
```

would produce the value of `x` first with the default mode, then with a decimal point for English, and a decimal comma for French.

If the decimal edit mode is `comma` during list-directed or namelist input/output, a semicolon acts as a value separator instead of a comma.

11.6 Defined derived-type input/output

Defined derived-type input/output allows the programmer to provide formatting specially tailored to a type and to transfer structures with pointer, allocatable, or private components.

Thus, it may be arranged that, when a derived-type object is encountered in an input/output list, a Fortran subroutine is called. This either reads some data from the file and constructs a value of the derived type or accepts a value of the derived type and writes some data to the file.

For formatted input/output, defined input/output for a type is active always for list-directed and namelist input/output, but for an explicit format specification, only when the `dt` data edit descriptor is used. For unformatted input/output, defined input/output for a type is always active.

Note that defined input/output fundamentally affects the way that I/O list items of derived type are expanded into their components. For formatted input/output, an item that will be processed by defined input/output is not expanded into its components (note that for an explicit format specification, this depends on the edit descriptor being `dt`; if it is not, the item is expanded as usual).

For unformatted input/output, a derived-type I/O list item for which a subcomponent will be processed by defined input/output is expanded into its components, unlike the usual case where the whole derived-type I/O list item is output as a single item. If a derived type has internal padding for alignment purposes, this padding will therefore not appear when part of it will be output by defined input/output.

The `dt` edit descriptor for formatted input/output provides an optional character string and an optional integer array to control the action. An example is

```
dt 'linked-list' (10, -4, 2)
```

If the character string is omitted, the effect is the same as if a string of length zero had been given. If the parenthetical list of integers is omitted, an array of size zero is passed. Note that if the parentheses appear, at least one integer must be specified.

Subroutines for defined derived-type I/O may be bound to the type as generic bindings (see Section 15.8.2) of the forms

```
generic :: read(formatted) => r1, r2
generic :: read(unformatted) => r3, r4, r5
generic :: write(formatted) => w1
generic :: write(unformatted) => w2, w3
```

which makes them accessible wherever an object of the type is accessible. An alternative is an interface block such as

```
interface read(formatted)
   module procedure r1, r2
end interface
```

The form of such a subroutine depends on whether it is for formatted or unformatted I/O:

```
subroutine formatted_io(dtv,unit,iotype,v_list,iostat,iomsg)
subroutine unformatted_io(dtv,unit,             iostat,iomsg)
```

`dtv` is a scalar of the derived type. It may be polymorphic (so that it can be called for the type or any extension of it, see Section 15.3.1). All length type parameters must be assumed. For output, it is of intent `in` and holds the value to be written. For input, it is of intent `inout` and is altered in accord with the values read.

unit is a scalar of intent `in` and type default `integer`. Its value is the unit on which input/output is taking place or negative if on an internal file.

iotype is a scalar of intent `in` and type `character(*)`. Its value is `'LISTDIRECTED'`, `'NAMELIST'`, or `'DT'//`*string*, where *string* is the character string from the `dt` edit descriptor.

v_list is a rank-one assumed-shape array of intent `in` and type default `integer`. Its value comes from the parenthetical list of the edit descriptor.

iostat is a scalar of intent `out` and type default `integer`. If an error condition occurs, it is given a positive value. Otherwise, if an end-of-file or end-of-record condition occurs it is given, respectively, the value `iostat_end` or `iostat_eor` of the intrinsic module `iso_fortran_env` (see Section 9.23). Otherwise, it must be given the value zero.

iomsg is a scalar of intent `inout` and type `character(*)`. If `iostat` is given a nonzero value, `iomsg` must be set to an explanatory message. Otherwise, it must not be altered.

The names of the subroutine and its arguments are not significant when they are invoked as part of input/output processing.

Within the subroutine, input/output to external files is limited to the specified unit and in the specified direction. Such a data transfer statement is called a **child** data transfer statement and the original statement is called the **parent**. No file positioning takes place at the beginning or end of the execution of a child data transfer statement. Any `advance=` specifier is ignored. I/O to an internal file is permitted. An I/O list may include a `dt` edit descriptor for a component of the `dtv` argument, with the obvious meaning. Execution of any of the statements `open`, `close`, `backspace`, `endfile`, and `rewind` is not permitted. Also, the procedure must not alter any aspect of the parent I/O statement, except through the `dtv` argument.

The file position on entry is treated as a left tab limit and there is no record termination on return. Therefore, positioning with `rec=` (for a direct-access file, Section 10.13) or `pos=` (for stream access, Section 10.16) is not permitted in a child data transfer statement.

This feature is not available in combination with asynchronous input/output (Section 10.15).

A simple example of derived-type formatted output follows. The derived-type variable `chairman` has two components. The type and an associated write-formatted procedure are defined in a module called `person_module` and might be invoked as shown in Figure 11.1.

The module that implements this is shown in Figure 11.2. From the edit descriptor `dt(15,6)`, it constructs the format `(a15,i 6)` in the local character variable `pfmt` and applies it. It would also be possible to check that `iotype` indeed has the value `'DT'` and to set `iostat` and `iomsg` accordingly.

Figure 11.1 A program with a `dt` edit descriptor.

```
program
  use person_module
  integer id, members
  type (person) :: chairman
  ⋮
  write (6, fmt="(i2, dt(15,6), i5)" ) id, chairman, members
! This writes a record with four fields, with lengths 2, 15, 6, 5,
! respectively
end program
```

Figure 11.2 A module containing a `write(formatted)` subroutine.

```
module person_module
   type :: person
      character (len=20) :: name
      integer :: age
   contains
      procedure :: pwf
      generic   :: write(formatted) => pwf
   end type person
contains
   subroutine pwf (dtv, unit, iotype, vlist, iostat, iomsg)
   ! Arguments
      class(person), intent(in)        :: dtv
      integer, intent(in)              :: unit
      character (len=*), intent(in)    :: iotype
      integer, intent(in)              :: vlist(:)
      ! vlist(1) and (2) are to be used as the field widths
      ! of the two components of the derived type variable.
      integer, intent(out)             :: iostat
      character (len=*), intent(inout) :: iomsg
      ! Local variable
      character (len=9) :: pfmt
      ! Set up the format to be used for output
      write (pfmt, '(a,i2,a,i2,a)' ) &
         '(a', vlist(1), ',i', vlist(2), ')'
      ! Now the child output statement
      write (unit, fmt=pfmt, iostat=iostat) dtv%name, dtv%age
   end subroutine pwf
end module person_module
```

In the Figure 11.3 example we illustrate the output of a structure with a pointer component and show a child data transfer statement itself invoking derived-type input/output. Here, we show the case where the same (recursive) subroutine is invoked in both cases. The variables of the derived type `node` form a chain, with a single value at each node and terminating with a null pointer. The subroutine `pwf` is used to write the values in the list, one per line.

Figure 11.3 A module containing a recursive `write(formatted)` subroutine.

```
module list_module
   type node
      integer              :: value = 0
      type (node), pointer :: next_node => null ( )
   contains
      procedure :: pwf
      generic   :: write(formatted) => pwf
   end type node
contains
   recursive subroutine pwf (dtv, unit, iotype, vlist, iostat, iomsg)
 ! Write the chain of values, each on a separate line in I9 format.
      class(node), intent(in)          :: dtv
      integer, intent(in)              :: unit
      character (len=*), intent(in)    :: iotype
      integer, intent(in)              :: vlist(:)
      integer, intent(out)             :: iostat
      character (len=*), intent(inout) :: iomsg
      write (unit, '(i9,/)', iostat = iostat) dtv%value
      if (iostat/=0) return
      if (associated(dtv%next_node)) &
         write (unit, '(dt)', iostat=iostat) dtv%next_node
   end subroutine pwf
end module list_module
```

11.7 Recursive input/output

A recursive input/output statement is one that is executed while another input/output statement is in execution. We met this in connection with derived-type input/output (Section 11.6); a child data transfer statement is recursive since it always executes while its parent is in execution. Recursive input/output is allowed for input/output to/from an internal file where the statement does not modify any internal file other than its own.

Recursive input/output is also permitted for external files, provided only that the same unit is not involved in both input/output actions. This is particularly useful while debugging and also for logging and error reporting. For example,

```
  print *, invert_matrix(x)
  ⋮
contains
  function invert_matrix(a)
    ⋮
    if (singular) then
      write (error_unit,*) &
        'Cannot invert singular matrix - continuing!'
      return
    end if
    ⋮
```

11.8 Summary

This chapter has described in full detail the edit descriptors used for formatting in I/O statements.

Exercises

1. Write suitable `print` statements to print the name and contents of each of the following arrays:

i) `real :: grid(10,10)`, ten elements to a line (assuming the values are between 1.0 and 100.0);

ii) `integer :: list(50)`, the odd elements only;

iii) `character(len=10) :: titles(20)`, two elements to a line;

iv) `real :: power(10)`, five elements to a line in engineering notation;

v) `logical :: flags(10)`, on one line;

vi) `complex :: plane(5)`, on one line.

2. Write separate list-directed input statements to fill each of the arrays of Exercise 1. For each statement write a sample first input record.

3. Write a function that formats a real input value, of a kind that has a decimal precision of 15 or more, in a suitable form for display as a monetary value in Euros. If the magnitude of the value is such that the 'cent' field is beyond the decimal precision, a string consisting of all asterisks should be returned.

4. Write a program that displays the effects of the `sign=` specifier and the `ss`, `sp`, and `s` edit descriptors. What output would you expect if the file is opened with `sign='suppress'`?

12. Operations on external files

12.1 Introduction

So far we have discussed the topic of external files in a rather superficial way. In the examples of the various I/O statements in the previous chapter, an implicit assumption has always been made that the specified file was actually available, and that records could be written to it and read from it. For sequential files, the file control statements described in the next section further assume that it can be positioned. In fact, these assumptions are not necessarily valid. In order to define explicitly and to test the status of external files, three file status statements are provided: `open`, `close`, and `inquire`. Before beginning their description, however, two new definitions are required.

A computer system contains, among other components, a CPU and a storage system. Modern storage systems are usually based on some form of disk, which is used to store files for long or short periods of time. The execution of a computer program is, by comparison, a transient event. A file may exist for years, whereas programs run for only seconds or minutes. In Fortran terminology, a file is said to *exist* not in the sense we have just used, but in the restricted sense that it exists as a file *to which the program might have access.* In other words, if the program is prohibited from using the file because of a password protection system, or because some other necessary action has not been taken, the file 'does not exist'.

A file which exists for a running program may be empty and may or may not be *connected* to that program. The file is connected if it is associated with a unit number known to the program. Such connection is usually made by executing an `open` statement for the file, but many computer systems will *preconnect* certain files which any program may be expected to use, such as terminal input and output. Thus, we see that a file may exist but not be connected. It may also be connected but not exist. This can happen for a preconnected new file. The file will only come into existence (be *created*) if some other action is taken on the file: executing an `open`, `write`, `print`, or `endfile` statement. A unit must not be connected to more than one file at once, and a file must not be connected to more than one unit at once.[1]

There are a number of other points to note with respect to files.

- The set of allowed names for a file is processor dependent.
- Both sequential and direct access may be available for some files, but normally a file is limited to sequential, direct, or stream access.

[1] But see Section 23.15.5, where Fortran 2018 lifts this restriction.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

- A file never contains both formatted and unformatted records.

Finally, we note that no statement described in this chapter applies to internal files.

12.2 Positioning statements for sequential files

When reading or writing an external file that is connected for sequential access, whether formatted or unformatted, it is sometimes necessary to perform other control functions on the file in addition to input and output. In particular, one may wish to alter the current position, which may be within a record, between records, ahead of the first record (at the *initial point*), or after the last record (at its *terminal point*). The following three statements are provided for these purposes.

12.2.1 The backspace statement

It can happen in a program that a series of records is being written and that, for some reason, the last record written should be overwritten by a new one. Similarly, when reading records, it may be necessary to reread the last record read, or to check-read a record which has just been written. For this purpose, Fortran provides the `backspace` statement, which has the syntax

```
backspace u
```

or

```
backspace ([unit=]u [,iostat=ios] [,err=error-label])
```

where *u* is a scalar integer expression whose value is the unit number, and the other optional specifiers have the same meaning as for a `read` statement. Again, keyword specifiers may be in any order, but the unit specifier must come first as a positional specifier.

The action of this statement is to position the file before the current record if it is positioned within a record, or before the preceding record if it is positioned between records. An attempt to backspace when already positioned at the beginning of a file results in no change in the file's position. If the file is positioned after an endfile record (Section 12.2.3), it becomes positioned before that record. It is not possible to backspace a file that does not exist, nor to backspace over a record written by a list-directed or namelist output statement (Sections 10.9 and 10.10). A series of `backspace` statements will backspace over the corresponding number of records. This statement is often very costly in computer resources and should be used as little as possible.

12.2.2 The rewind statement

In an analogous fashion to rereading, rewriting, or check-reading a record, a similar operation may be carried out on a complete file. For this purpose the `rewind` statement,

```
rewind u
```

or

```
rewind ([unit=]u [,iostat=ios] [,err=error-label])
```

may be used to reposition a file, whose unit number is specified by the scalar integer expression *u*. Again, keyword specifiers may be in any order, but the unit specifier must come first as a positional specifier. If the file is already at its beginning, there is no change in its position. The statement is permitted for a file that does not exist, and has no effect.

12.2.3 The endfile statement

The end of a file connected for sequential access is normally marked by a special record which is identified as such by the computer hardware, and computer systems ensure that all files written by a program are correctly terminated by such an **endfile record**. In doubtful situations, or when a subsequent program step will reread the file, it is possible to write an endfile record explicitly using the `endfile` statement:

```
endfile u
```

or

```
endfile ([unit=]u [,iostat=ios] [,err=error-label])
```

where *u*, once again, is a scalar integer expression specifying the unit number. Again, keyword specifiers may be in any order, but the unit specifier must come first as a positional specifier. The file is then positioned after the endfile record. This endfile record, if subsequently read by a program, must be handled using the `iostat=`*ios* or `end=`*end-label* specifier of the `read` statement, otherwise program execution will terminate. Prior to data transfer, a file must not be positioned after an endfile record, but it is possible to backspace or rewind across an endfile record, which allows further data transfer to occur. An endfile record is written automatically whenever either a backspace or rewind operation follows a write operation as the next operation on the unit, or the file is closed by execution of a `close` statement (Section 12.5), by an `open` statement for the same unit (Section 12.4), or by normal program termination.

If the file may also be connected for direct access, only the records ahead of the endfile record are considered to have been written and only these may be read during a subsequent direct-access connection.

Note that if a file is connected to a unit but does not exist for the program, it will be made to exist by executing an `endfile` statement on the unit.

12.2.4 Data transfer statements

Execution of a data transfer statement (`read`, `write`, or `print`) for a sequential file also affects the file position. If it is between records, it is moved to the start of the next record unless non-advancing access is in operation. Data transfer then takes place, which usually moves the position. No further movement occurs for non-advancing access. For advancing access, the position finally moves to follow the last record transferred.

12.3 The flush statement

Execution of a `flush` statement for an external file causes data written to it to be available to other processes, or causes data placed in it by means other than Fortran to be available to a

`read` statement. The syntax is just like that of the file positioning statements (Section 12.2), for instance

```
flush(6)
```

In combination with `advance='no'` or stream access (Section 10.16), it permits the program to ensure that data written to one unit are sent to the file before requesting input on another unit; that is, that 'prompts' appear promptly.

12.4 The open statement

The `open` statement is used to connect an external file to a unit, create a file that is preconnected, create a file and connect it to a unit, or change certain properties of a connection. The syntax is

`open (`*[*`unit=`*]u [,olist]*`)`

where *u* is a scalar integer expression specifying the external file unit number, and *olist* is a list of optional specifiers. If the unit is specified with `unit=`, it may appear in *olist*. A specifier must not appear more than once. In the specifiers, all entities are scalar and all characters are of default kind. In character expressions, any trailing blanks are ignored and, except for `file=`, any upper-case letters are converted to lower case. The specifiers are as follows:

`access=` *acc*, where *acc* is a character expression that provides one of the values `sequential`, `direct`, or `stream`. For a file which already exists, this value must be an allowed value. If the file does not already exist, it will be brought into existence with the appropriate access method. If this specifier is omitted, the value `sequential` will be assumed.

`action=` *act*, where *act* is a character expression that provides the value `read`, `write`, or `readwrite`. If `read` is specified, the `write`, `print` and `endfile` statements must not be used for this connection; if `write` is specified, the `read` statement must not be used (and `backspace` and `position='append'` may fail on some systems); if `readwrite` is specified, there is no restriction. If the specifier is omitted, the default value is processor dependent.

`asynchronous=` *asy*, where *asy* is a character expression that provides the value `yes` or `no`. If `yes` is specified, asynchronous input/output on the unit is allowed. If `no` is specified, asynchronous input/output on the unit is not allowed. If the specifier is omitted, the default value is `no`.

`blank=` *bl*, where *bl* is a character expression that provides the value `null` or `zero`. This connection must be for formatted I/O. This specifier sets the default for the interpretation of blanks in numeric input fields, as discussed in the description of the `bn` and `bz` edit descriptors (Section 11.5.1). If the value is `null`, such blanks will be ignored (except that a completely blank field is interpreted as zero). If the value is `zero`, such blanks will be interpreted as zeros. If the specifier is omitted, the default is `null`.

decimal= *decimal*, where *decimal* is a character expression that specifies the decimal edit mode (see Section 11.5.4).

delim= *del*, where *del* is a character expression that provides the value `quote`, `apostrophe`, or `none`. If `apostrophe` or `quote` is specified, the corresponding character will be used to delimit character constants written with list-directed or `namelist` formatting, and it will be doubled where it appears within such a character constant; also, non-default character values will be preceded by kind values. No delimiting character is used if `none` is specified, nor does any doubling take place. The default value if the specifier is omitted is `none`. This specifier may appear only for formatted files.

encoding= *enc*, where *enc* is a character expression that provides the value `utf-8` or `default`. If `utf-8` is specified, UTF-8 coding is used for the file (see Section 10.14). If `default` is specified, the encoding is processor dependent.

err= *error-label*, where *error-label* is the label of a statement in the same scoping unit to which control will be transferred in the event of an error occurring during execution of the statement.

file= *fln*, where *fln* is a character expression that provides the name of the file. If this specifier is omitted and the unit is not connected to a file, the `status=` specifier must be specified with the value `scratch` and the file connected to the unit will then depend on the computer system. Whether the interpretation is case sensitive varies from system to system.

form= *fm*, where *fm* is a character expression that provides the value `formatted` or `unformatted`, and determines whether the file is to be connected for formatted or unformatted I/O. For a file which already exists, the value must be an allowed value. If the file does not already exist, it will be brought into existence with an allowed set of forms that includes the specified form. If this specifier is omitted, the default is `formatted` for sequential access and `unformatted` for direct or stream access.

iostat= *ios*, where *ios* is a default integer variable which is set to zero if the statement is correctly executed, and to a positive value otherwise.

newunit= *nu*, where *nu* is an integer variable whose value is a unique negative unit number on a successful `open`; the processor will choose a value that does not clash with anything it is using internally. To avoid any confusion in the result of the `number=` specifier of the `inquire` statement, where -1 indicates a file that is not connected, `newunit=` will never return -1.

pad= *pad*, where *pad* is a character expression that provides the value `yes` or `no`. If `yes` is specified, a formatted input record will be regarded as padded out with blanks whenever an input list and the associated format specify more data than appear in the record. (If `no` is specified, the length of the input record must not be less than that specified by the input list and the associated format, except in the presence of an `advance='no'` specifier and either an `eor=` or an `iostat=` specification.) The default value if the

specifier is omitted is `yes`. For non-default characters, the blank padding character is processor dependent.

`position=` *pos*, where *pos* is a character expression that provides the value `asis`, `rewind`, or `append`. The access method must be sequential, and if the specifier is omitted the default value `asis` will be assumed. A new file is positioned at its initial point. If `asis` is specified and the file exists and is already connected, the file is opened without changing its position; if `rewind` is specified, the file is positioned at its initial point; if `append` is specified and the file exists, it is positioned ahead of the endfile record if it has one (and otherwise at its terminal point). For a file which exists but is not connected, the effect of the `asis` specifier on the file's position is unspecified.

`recl=` *rl*, where *rl* is an integer expression whose value must be positive. For a direct-access file, it specifies the length of the records, and is obligatory. For a sequential file, it specifies the maximum length of a record and is optional, with a default value that is processor dependent. It must not appear for a stream-access file. For formatted files, the length is the number of characters for records that contain only default characters; for unformatted files it is system dependent but the `inquire` statement (end of Section 12.6) may be used to find the length of an I/O list. In either case, for a file which already exists, the value specified must be allowed for that file. If the file does not already exist, the file will be brought into existence with an allowed set of record lengths that includes the specified value.

`round=` *rnd*, where *rnd* is a character expression that specifies the rounding mode (see Section 11.5.2).

`sign=` *sign*, where *sign* is a character expression that specifies the sign mode (see Section 11.5.3).

`status=` *st*, where *st* is a character expression that provides the value `old`, `new`, `replace`, `scratch`, or `unknown`. The `file=` specifier must be present if `new` or `replace` is specified or if `old` is specified and the unit is not connected; the `file=` specifier must not be present if `scratch` is specified. If `old` is specified, the file must already exist; if `new` is specified, the file must not already exist, but will be brought into existence by the action of the `open` statement. The status of the file then becomes `old`. If `replace` is specified and the file does not already exist, the file is created; if the file does exist, the file is deleted and a new file is created with the same name. In each case the status is changed to `old`. If the value `scratch` is specified, the file is created and becomes connected, but it cannot be kept after completion of the program or execution of a `close` statement (Section 12.5). If `unknown` is specified, the status of the file is system dependent. This is the default value of the specifier, if it is omitted.

An example of an `open` statement is

```
open (2, iostat=ios, err=99, file='cities',                    &
      status='new', access='direct', recl=100)
```

which brings into existence a new, direct-access, unformatted file named `cities`, whose records have length 100. The file is connected to unit number 2. Failure to execute the

statement correctly will cause control to be passed to the statement labelled 99, where the value of `ios` may be tested.

The `open` statements in a program are best collected together in one place, so that any changes which might have to be made to them when transporting the program from one system to another can be carried out without having to search for them. Regardless of where they appear, the connection may be referenced in any program unit of the program.

The purpose of the `open` statement is to connect a file to a unit. If the unit is, however, already connected to a file then the action may be different. If the `file=` specifier is omitted, the default is the name of the connected file. If the file in question does not exist, but is preconnected to the unit, then all the properties specified by the `open` statement become part of the connection. If the file is already connected to the unit, then of the existing attributes only the `blank=`, `decimal=`, `delim=`, `pad=`, `round=`, `sign=`, `err=`, and `iostat=` specifiers may have values different from those already in effect. If the unit is already connected to another file, the effect of the `open` statement includes the action of a prior `close` statement on the unit (without a `status=` specifier, see next section).

A file already connected to one unit must not be specified for connection to another unit.

In general, by repeated execution of the `open` statement on the same unit, it is possible to process in sequence an arbitrarily high number of files, whether they exist or not, as long as the restrictions just noted are observed.

12.5 The close statement

The purpose of the `close` statement is to disconnect a file from a unit. Its form is

`close (`*`[`*`unit=`*`]u [,`*`iostat=`*`ios] [,`*`err=`*`error-label] [,`*`status=`*`st]`*`)`

where *u, ios,* and *error-label* have the same meanings as described in the previous section for the `open` statement. Again, keyword specifiers may be in any order, but the unit specifier must come first as a positional specifier.

The function of the `status=` specifier is to determine what will happen to the file once it is disconnected. The value of *st*, which is a scalar default character expression, may be either `keep` or `delete`, ignoring any trailing blanks and converting any upper-case letters to lower case. If the value is `keep`, a file that exists continues to exist after execution of the `close` statement, and may later be connected again to a unit. If the value is `delete`, the file no longer exists after execution of the statement. In either case, the unit is free to be connected again to a file. The `close` statement may appear anywhere in the program, and if executed for a non-existing or unconnected unit, acts as a 'do nothing' statement. The value `keep` must not be specified for files with the status `scratch`.

If the `status=` specifier is omitted, its default value is `keep` unless the file has status `scratch`, in which case the default value is `delete`. On normal termination of execution, all connected units are closed, as if `close` statements with omitted `status=` specifiers were executed.

An example of a `close` statement is

```
close (2, iostat=ios, err=99, status='delete')
```

12.6 The inquire statement

The status of a file can be defined by the operating system prior to execution of the program, or by the program itself during execution, either by an `open` statement or by some action on a preconnected file which brings it into existence. At any time during the execution of a program it is possible to inquire about the status and attributes of a file using the `inquire` statement. Using a variant of this statement, it is similarly possible to determine the status of a unit, for instance whether the unit number exists for that system (that is, whether it is an allowed unit number), whether the unit number has a file connected to it, and, if so, which attributes that file has. Another variant permits an inquiry about the length of an output list when used to write an unformatted record.

Some of the attributes that may be determined by use of the `inquire` statement are dependent on others. For instance, if a file is not connected to a unit, it is not meaningful to inquire about the form being used for that file. If this is nevertheless attempted, the relevant specifier is undefined.

The three variants are known as `inquire` by file, `inquire` by unit, and `inquire` by output list. In the description of the `inquire` statement which follows, the first two variants will be described together. Their forms are

```
inquire (unit=u, ilist)
```

for `inquire` by unit, where *u* is a scalar integer expression specifying an external unit, and

```
inquire ( file=fln, ilist)
```

for `inquire` by file, where *fln* is a scalar character expression whose value, ignoring any trailing blanks, provides the name of the file concerned. Whether the interpretation is case sensitive is system dependent. If the unit or file is specified by keyword, it may appear in *ilist*. A specifier must not occur more than once in the list of optional specifiers, *ilist*. All assignments occur following the usual rules, and all values of type character, apart from that for the `name=` specifier, are in upper case. The specifiers, in which all variables are scalar and those of integer or logical type may be of of any kind, are as follows:

`access=` *acc*, where *acc* is a character variable that is assigned one of the values `SEQUENTIAL`, `DIRECT`, or `STREAM` depending on the access method for a file that is connected, and `UNDEFINED` if there is no connection.

`action=` *act*, where *act* is a character variable that is assigned the value `READ`, `WRITE`, or `READWRITE`, according to the connection. If there is no connection, the value assigned is `UNDEFINED`.

`asynchronous=` *asynch*, where *asynch* is a character variable that is assigned the value `YES` if the file is connected and asynchronous input/output on the unit is allowed; it is assigned the value `NO` if the file is connected and asynchronous input/output on the unit is not allowed. If there is no connection, it is assigned the value `UNDEFINED`.

`blank=` *bl*, where *bl* is a character variable that is assigned the value `NULL` or `ZERO`, depending on whether the blanks in numeric fields are by default to be interpreted as null fields or zeros, respectively, and `UNDEFINED` if there is either no connection, or if the connection is not for formatted I/O.

decimal= *dec*, where *dec* is a character variable that is assigned the value `COMMA` or `POINT`, corresponding to the decimal edit mode in effect for a connection for formatted input/output. If there is no connection, or if the connection is not for formatted input/output, it is assigned the value `UNDEFINED`.

delim= *del*, where *del* is a character variable that is assigned the value `QUOTE`, `APOSTROPHE`, or `NONE`, as specified by the corresponding `open` statement (or by default). If there is no connection, or if the file is not connected for formatted I/O, the value assigned is `UNDEFINED`.

direct= *dir*, **sequential=** *seq*, and **stream=** *stm*, where *dir*, *seq*, and *stm* are character variables that are assigned the value `YES`, `NO`, or `UNKNOWN`, depending on whether the file *may* be opened for sequential, direct, or stream access, respectively, or whether this cannot be determined.

encoding= *enc*, where *enc* is a character variable that is assigned the value `utf-8` or `default` corresponding to the encoding in effect.

err= *error-label* and **iostat=** *ios* have the meanings described for them in the `open` statement in Section 12.4. The `iostat=` variable is the only one which is defined if an error condition occurs during the execution of the statement.

exist= *ex*, where *ex* is a logical variable. The value true is assigned to *ex* if the file (or unit) exists, and false otherwise.

form= *frm*, where *frm* is a character variable that is assigned one of the values `FORMATTED` or `UNFORMATTED`, depending on the form for which the file is actually connected, and `UNDEFINED` if there is no connection.

formatted= *fmt* and **unformatted=** *unf*, where *fmt* and *unf* are character variables that are assigned the value `YES`, `NO`, or `UNKNOWN`, depending on whether the file *may* be opened for formatted or unformatted access, respectively, or whether this cannot be determined.

id= *id*, where the integer variable `id` identifies a particular asynchronous input/output operation (see **pending=**).

iomsg= *message*, where *message* identifies a scalar variable of type default character into which the processor places a message if an error, end-of-file, or end-of-record condition occurs during execution of the statement (see also Section 10.7).

named= *nmd* and **name=** *nam*, where *nmd* is a logical variable that is assigned the value true if the file has a name, and false otherwise. If the file has a name, the character variable *nam* will be assigned the name. This value is not necessarily the same as that given in the `file` specifier, if used, but may be qualified in some way. However, in all cases it is a name which is valid for use in a subsequent `open` statement, and so `inquire` can be used to determine the actual name of a file before connecting it. Whether the file name is case sensitive is system dependent.

nextrec= *nr*, where *nr* is an integer variable that is assigned the value of the number of the last record read or written, plus one. If no record has yet been read or written, it is assigned the value 1. If the file is not connected for direct access or if the position is indeterminate because of a previous error, *nr* becomes undefined.

number= *num*, where *num* is an integer variable that is assigned the value of the unit number connected to the file, or −1 if no unit is connected to the file.

opened= *open*, where *open* is a logical variable. The value true is assigned to *open* if the file (or unit) is connected to a unit (or file), and false otherwise.

pad= *pad*, where *pad* is a character variable that is assigned the value `YES` or `NO`, as specified by the corresponding `open` statement (or by default). If there is no connection, or if the file is not connected for formatted I/O, the value assigned is `UNDEFINED`.

pending= *pending*, where, if an `id=` specifier is present, the scalar default logical variable *pending* is given the value true if the particular input/output operation is still pending and false otherwise (see also Section 10.15).

pos= *pos*, where *pos* is a scalar integer variable that is assigned the current position in a stream-access file (see also Section 10.16).

position= *pos*, where *pos* is a character variable that is assigned the value `REWIND`, `APPEND`, or `ASIS`, as specified in the corresponding `open` statement, if the file has not been repositioned since it was opened. If there is no connection, or if the file is connected for direct access, the value is `UNDEFINED`. If the file has been repositioned since the connection was established, the value is processor dependent (but must not be `REWIND` or `APPEND` unless that corresponds to the true position).

read= *rd*, where *rd* is a character variable that is assigned the value `YES`, `NO`, or `UNKNOWN`, according to whether read is allowed, not allowed, or is undetermined for the file.

readwrite= *rw*, where *rw* is a character variable that is assigned the value `YES`, `NO`, or `UNKNOWN`, according to whether read/write is allowed, not allowed, or is undetermined for the file.

recl= *rec*, where *rec* is an integer variable that is assigned the value of the record length of a file connected for direct access, or the maximum record length allowed for a file connected for sequential access. The length is the number of characters for formatted records containing only characters of default type, and system dependent otherwise. If there is no connection or the file is connected for stream access, *rec* becomes undefined.

round= *rnd*, where *rnd* is a character variable that is assigned the value `UP`, `DOWN`, `ZERO`, `NEAREST`, `COMPATIBLE`, or `PROCESSOR_DEFINED` according to the rounding mode in effect (see Section 11.5.2). If there is no connection, or if the file is not connected for formatted I/O, the value assigned is `UNDEFINED`.

sign= *sign*, where *sign* is a character variable that is assigned the value PLUS, SUPPRESS, or PROCESSOR_DEFINED, according to the sign mode in effect (see Section 11.5.3). If there is no connection, or if the file is not connected for formatted I/O, the value assigned is UNDEFINED.

size= *size*, where *size* is an integer variable that is assigned the size of the file in file storage units. If the file size cannot be determined, the variable is assigned the value −1. For a file that may be connected for stream access, the file size is the number of the highest-numbered file storage unit in the file. For a file that may be connected for sequential or direct access, the file size may be different from the number of storage units implied by the data in the records; the exact relationship is processor dependent.

write= *wr*, where *wr* is a character variable that is assigned the value YES, NO, or UNKNOWN, according to whether write is allowed, not allowed, or is undetermined for the file.

A variable that is a specifier on an inquire statement or is associated with one must not appear in another specifier in the same statement.

The third variant of the inquire statement, inquire by I/O list, has the form

```
inquire (iolength=length) olist
```

where *length* is a scalar integer variable of default kind and is used to determine the length of an unformatted output list in processor-dependent units, and might be used to establish whether, for instance, an output list is too long for the record length given in the recl= specifier of an open statement, or be used as the value of the length to be supplied to a recl= specifier (see Figure 10.5 in Section 10.13).

An example of the inquire statement, for the file opened as an example of the open statement in Section 12.4, is

```
logical            :: ex, op
character (len=11) :: nam, acc, seq, frm
integer            :: irec, nr
inquire (2, err=99, exist=ex, opened=op, name=nam, access=acc, &
    sequential=seq, form=frm, recl=irec, nextrec=nr)
```

After successful execution of this statement, the variables provided will have been assigned the following values:

```
ex      .true.
op      .true.
nam      cities bbbbb
acc      DIRECT bbbbb
seq      NO bbbbbbbbb
frm      UNFORMATTED
irec     100
nr       1
```

(assuming no intervening read or write operations).

The three I/O status statements just described are perhaps the most indigestible of all Fortran statements. They provide, however, a powerful and portable facility for the dynamic allocation and deallocation of files, completely under program control, which is far in advance of that found in any other programming language suitable for scientific applications.

12.7 Summary

This chapter has completed the description of the input/output features begun in the previous two chapters, and together they provide a complete reference to all the facilities available.

Exercises

1. A direct-access file is to contain a list of names and initials, to each of which there corresponds a telephone number. Write a program which opens a sequential file and a direct-access file, and copies the list from the sequential file to the direct-access file, closing it for use in another program. Write a second program which reads an input record containing either a name or a telephone number (from a terminal if possible), and prints out the corresponding entry (or entries) in the direct-access file if present, and an error message otherwise. Remember that names are as diverse as Wu, O'Hara, and Trevington-Smythe, and that it is insulting for a computer program to corrupt or abbreviate people's names. The format of the telephone numbers should correspond to your local numbers, but the actual format used should be readily modifiable to another.

13. Advanced type parameter features

The advanced type parameter features consist of type parameter inquiry and the ability to parameterize derived types.

13.1 Type parameter inquiry

The (current) value of a type parameter of a variable can be discovered by a **type parameter inquiry**. This uses the same syntax as for component access, but the value is always scalar, even if the object is an array; for example, in

```
real(selected_real_kind(10,20)) :: z(100)
⋮
print *,z%kind
```

a single value is printed, that being the result of executing the reference to the intrinsic function `selected_real_kind`. This particular case is equivalent to `kind(z)`. However, the type parameter inquiry may be used even when the intrinsic function is not available; for example, in

```
subroutine write_centered(ch, len)
   character(*), intent(inout) :: ch
   integer, intent(in)         :: len
   integer                     :: i
   do i=1, (len-ch%len)/2
```

it would not be possible to replace the type parameter inquiry `ch%len` with a reference to the intrinsic function `len(ch)` because `len` is the name of a dummy argument.

Note that this syntax must not be used to alter the value of a type parameter, say by appearing on the left-hand side of an assignment statement.

13.2 Parameterized derived types

A derived type can have type parameters, in exact analogy with type parameters of intrinsic types. Like intrinsic type parameters, derived type parameters come in two flavours: those that must be known at compile time (like the `kind` parameter for type `real`), and those whose evaluation may be deferred until run time (like the `len` parameter for type `character`). The former are known as **kind** type parameters (because, for the intrinsic types, these are all named `kind`), and the latter as **length** type parameters (by analogy with character length).

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

13.2.1 Defining a parameterized derived type

To define a derived type that has type parameters, the type parameters are listed on the type definition statement and must also be explicitly declared at the beginning of the derived-type definition. For example,

```
type matrix(real_kind, n, m)
   integer, kind   :: real_kind
   integer, len    :: n, m
   real(real_kind) :: value(n, m)
end type matrix
```

defines a derived type `matrix` with one kind type parameter named `real_kind` and two length type parameters named `n` and `m`. All type parameters must be explicitly declared to be of type `integer` with the attribute `kind` or `len` to indicate a kind or length parameter, respectively. Within the derived-type definition a kind type parameter may be used in both constant and specification expressions, but a length type parameter may only be used in a specification expression (that is, for array bounds and for other length type parameters such as character length). There is, however, no requirement that a type parameter be used at all.

If a component is default-initialized, its type parameters and array bounds must be constant expressions. For example, if a component is declared as

```
character(n) :: ch(m) = 'xyz'
```

both `n` and `m` must be named constants or kind type parameters.

Examples of valid and invalid parameterized derived types are shown in Figure 13.1.

Figure 13.1 A valid and an invalid parameterized derived type.

```
type goodtype(p1, p2, p3, p4)
   integer, kind     :: p1, p3
   integer, len      :: p2, p4
   real(kind=p1)     :: c1      ! ok, p1 is a kind type parameter
   character(len=p2) :: c2      ! ok, this is a specification expr
   complex           :: c3(p3)  ! ok, p3 can be used anywhere
   integer           :: c4 = p1 ! ok, p1 can be used anywhere
   ! p4 has not been used, but that is ok.
end type goodtype

type badtype(p5)
   integer, len  :: p5
   real(kind=p5) :: x      ! Invalid, p5 is not a kind type parameter
   integer       :: y = p5 ! Invalid, p5 is not a kind type parameter
end type badtype
```

When declaring an entity of a parameterized derived type, its name is qualified by the type parameters in a type declaration statement of the form

`type(` *derived-type-spec* `)`

where *derived-type-spec* is

derived-type-name (*type-param-spec-list*)

in which *derived-type-name* is the name of the derived type and *type-param-spec* is

[keyword =] type-param-value

The keyword must be the name of one of the type parameters of the type. Like keyword arguments in procedure calls, after a *type-param-spec* that includes a *keyword* = clause, any further type parameter specifications must include a keyword. Note that this is consistent with the syntax for specifying type parameters for intrinsic types. Here are some examples for variables of our type `matrix`:

```
type(matrix(kind(0.0), 10, 20)) :: x
type(matrix(real_kind=kind(0d0), n=n1, m=n2)) :: y
```

13.2.2 Assumed and deferred type parameters

As for a dummy argument of the intrinsic type `character`, a length type parameter for a derived type dummy argument may be **assumed**. In this case, its value is indicated by a *type-param-value* that is an asterisk and is taken from that of the actual argument, as in the example:

```
subroutine print_matrix(z)
   type(matrix(selected_real_kind(30,999), n=*, m=*)) :: z
   ⋮
```

An asterisk may also be used for an assumed type parameter in the `allocate` statement (see Section 15.4) and the `select type` statement (see Section 15.7).

As for the intrinsic type `character`, a length *type-param-value* for a derived type may be **deferred**. For example, in

```
type(matrix(selected_real_kind(30,999), n=:, m=:)), pointer :: mp
type(matrix(selected_real_kind(30,999), n=100, m=200)), target :: x
mp => x
```

the values for `mp` of both `n` and `m` are deferred until association or allocation. After execution of the pointer assignment, the `n` and `m` type parameter values of `mp` are equal to those of `x` (100 and 200, respectively).

13.2.3 Default type parameter values

All type parameters for intrinsic types have default values. Similarly, a type parameter for a derived type may have a default value; this is declared using the same syntax as for default initialization of components, for example

```
type char_with_max_length(maxlen, kind)
   integer, len            :: maxlen = 255
   integer, kind           :: kind = selected_char_kind('default')
   integer                 :: len
   character(maxlen, kind) :: value
end type char_with_max_length
```

When declaring objects of type `char_with_max_length`, it is not necessary to specify the `kind` or `maxlen` parameters if the default values are acceptable.

This also illustrates that, in many simple cases that have only one kind type parameter, the natural name for the type parameter may be `kind` (just as it is for the intrinsic types). That name was chosen in this particular example because `char_with_max_length` was meant to be as similar to the intrinsic type `character` as possible. Note that this choice does not conflict with the attribute keyword `kind`.

13.2.4 Derived type parameter inquiry

The value of a type parameter of a variable can be discovered by a type parameter inquiry, as with intrinsic types (see Section 13.1). For example, using the type of Figure 13.2, in

```
type(character_with_max_length(...,...)) :: x, y(100)
⋮
print *,x%kind
print *,y%maxlen
```

the values of the `kind` type parameter of `x` and the `maxlen` type parameter of `y` will be printed.

Figure 13.2 Using default parameter values.

```
type character_with_max_length(maxlen, kind)
   integer, len    :: maxlen
   integer, kind   :: kind = selected_char_kind('default')
   integer         :: length = 0
   character(kind) :: value(maxlen)
end type character_with_max_length
⋮
type(character_with_max_length(100)) :: name
⋮
name = character_with_max_length(100)('John Hancock')
```

Because component syntax is used to access the value of a type parameter, a type is not allowed to have a component whose name is the same as one of the parameters of the type.

13.2.5 Structure constructor

In a structure constructor for a derived type that has type parameters, the type parameters are specified in parentheses immediately after the type name. Further, if the type parameters have default values, they may be omitted, as in the example in Figure 13.2.

Exercises

1. Write a replacement for the intrinsic type complex, which is opaque (has private components), uses polar representation internally, and has a single `kind` parameter that has the same default as the intrinsic type.
2. Write replacements for the character concatenation operator `(//)` and the intrinsic function `index` which work on type `char_with_max_length` (defined in Section 13.2.3).

14. Procedure pointers

Procedure pointers provide the ability to associate a pointer with a procedure, similar to the way dummy procedures become associated with actual procedures.

14.1 Abstract interfaces

We have seen that to declare a dummy or an external procedure with an explicit interface one needs to use an interface block. This is fine for a single procedure, but is somewhat verbose for declaring several procedures that have the same interface (apart from the procedure names). Also, there are several situations where this becomes impossible (procedure pointer components or abstract type-bound procedures). For these situations the **abstract interface** is available. An abstract interface gives a name to a set of characteristics and argument keyword names that would constitute an explicit interface to a procedure, without declaring any actual procedure to have those characteristics. This abstract interface name may be used in the `procedure` statement to declare procedures which might be external procedures, dummy procedures, procedure pointers, or deferred type-bound procedures.

An abstract interface block contains the `abstract` keyword, and each procedure body declared therein defines a new abstract interface. For example, given the abstract interface block

```
abstract interface
   subroutine boring_sub_with_no_args
   end subroutine boring_sub_with_no_args
   real function r2_to_r(a, b)
      real, intent(in) :: a, b
   end function r2_to_r
end interface
```

the declaration statements

```
procedure(boring_sub_with_no_args) :: sub1, sub2
procedure(r2_to_r) :: modulus, xyz
```

declare `sub1` and `sub2` to be subroutines with no actual arguments, and `modulus` and `xyz` to be `real` functions of two `real` arguments. The names `boring_sub_with_no_args` and `r2_to_r` are local to the scoping unit in which the abstract interface block is declared, and do not represent procedures or other global entities in their own right.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

As well as with abstract interfaces, the `procedure` statement may be used with any specific procedure that has an explicit interface. For example, if `fun` has an explicit interface,

```
procedure(fun) :: fun2
```

declares `fun2` to be a procedure with an identical interface to that of `fun`.

The `procedure` statement is not available for a set of generic procedures, but can be used for a specific procedure that is a member of a generic set. All the intrinsic procedures are generic, but a few also have specific versions that may be passed as an actual argument and are listed in Table B.2. An intrinsic may be named in a `procedure` statement only if the name appears in this table.

In addition, the `procedure` statement can be used to declare procedures that have implicit interfaces; instead of putting the name of a procedure inside the parentheses, either nothing or a type specification is used. For example,

```
procedure() x
procedure(real) y
procedure(complex(kind(0.0d0))) z
```

declares `x` to be a procedure (which might be a subroutine or an implicitly typed function[1]), `y` to be a `real` function, and `z` to be a (double) `complex` function. This is exactly equivalent to

```
external :: x
real, external :: y
complex(kind(0.0d0)), external :: z
```

For these cases the `procedure` statement offers no useful functionality over the `external` or type declaration statement; it really only comes into its own when declaring procedure pointers (see next section).

The full syntax of the `procedure` statement is

`procedure (` *[proc-interface]* `) [ [,` *proc-attr-spec]* `... :: ]` *proc-decl-list*

where a *proc-attr-spec* is one of

`bind (c [ , name=`*character-string]* `)`
`intent (`*inout*`)`
`optional`
`pointer`
`private`
`protected`
`public`
`save`

and a *proc-decl* is

procedure-name [=> procedure-pointer-init]

where *procedure-pointer-init* is a reference to the intrinsic function `null` with no arguments, or the name of a non-elemental external or module procedure.[2] (The `bind` attribute for procedures is described in Section 19.9.)

[1] The latter is not a possibility in a module or if `implicit none` is in effect.

[2] Or a specific intrinsic name, see Section B.3.6.

Each *proc-attr-spec* gives all the procedures declared in that statement the corresponding attribute. The `intent`, `protected`, and `save` attributes, and the initialization (to being a null pointer or associated with another procedure), may only appear if the procedures are pointers.

14.2 Procedure pointers

The pointers that were discussed in previous chapters were **data pointers** that can be associated with a data object. A **procedure pointer** is a pointer that can be associated with a procedure instead of a data object. It may have an explicit or implicit interface and its association with a target is as for a dummy procedure, so its interface is not permitted to be generic or elemental.

14.2.1 Named procedure pointers

A procedure pointer is declared by specifying that it is both a procedure and has the `pointer` attribute. For example,

```
pointer :: sp
interface
   subroutine sp(a, b)
      real, intent(inout) :: a
      real, intent(in)    :: b
   end subroutine sp
end interface
real, external, pointer :: fp
```

declares `sp` to be a pointer to a subroutine with the specified explicit interface and declares `fp` to be a pointer to a scalar `real` function with an implicit interface. More usually, a procedure pointer is declared with the `procedure` statement specifying the `pointer` attribute:

```
procedure(sp), pointer :: p1 ! Pointer with the interface of sp
procedure(), pointer :: p2 ! Pointer with an implicit interface
```

If a procedure pointer is currently associated (is neither disassociated nor undefined), its target may be invoked by referencing the pointer. For example,

```
fp => fun
sp => sub
print *, fp(x)  ! prints fun(x)
call sp(a, b)   ! calls sub
```

14.2.2 Procedure pointer components

A component of a derived type is permitted to be a procedure pointer. It must be declared using the `procedure` statement. For example, to define a type for representing a list of procedures (each with the same interface) to be called at some time, a procedure pointer component can be used, see Figure 14.1.

Figure 14.1 A type with a procedure pointer component.

```
type process_list
   procedure(process_interface), pointer :: process
   type(process_list), pointer           :: next => null()
end type process_list
abstract interface
   subroutine process_interface( ... )
      ⋮
   end subroutine process_interface
end interface
```

A procedure pointer component may be pointer-assigned to a procedure pointer, passed as an actual argument, or invoked directly. For example,

```
type(process_list) :: x, y(10)
procedure(process_interface), pointer :: p
⋮
p => x%process
call another_subroutine(x%process)
call y(i)%process(...)
```

Note that, just as with a data pointer component, in a reference to a procedure pointer component, the object of which the pointer is a component must be scalar (because there are no arrays of pointers in Fortran).

When a procedure is called through a pointer component of an object there is often a need to access the object itself; this is the topic of the next section.

14.2.3 The pass attribute

When a procedure pointer component (or a type-bound procedure, Section 15.8) is invoked, the object through which it is invoked is normally passed to the procedure as its first actual argument and the items in the parenthesized list are the other actual arguments. This could be undesirable; for instance, it might be wished to pass the object to a dummy argument other than the first, or not to pass it at all.

To pass the invoking object to a different dummy argument, the `pass` attribute is used. An example is shown in Figure 14.2. The dummy argument to which the object is to be passed is known as the *passed-object dummy argument*. Obviously, the dummy argument must be scalar and of the type of the object. Because the object is not required to be a pointer or allocatable, the dummy argument is required not to be a pointer or allocatable. Also, any length type parameters are required to be assumed and it is not permitted to have the `value` attribute.

Unless the type has the `sequence` (Appendix A.2) or `bind` (Section 19.4) attribute, or is `c_ptr` or `c_funptr`, it is extensible and the actual argument may be of an **extended type**. To

allow for this, the passed object dummy argument is required to be declared with the keyword `class` instead of `type`, see Figure 14.2. Type extension is fully discussed in Chapter 15.

Note that the `pass` attribute applies to the procedure pointer component and not to the procedure with which it is associated. For example, the procedure pointer might be associated from time to time with two different procedures; the object might be passed as the first argument in the first case and as the second argument in the second case. However, if the associated procedure is invoked through some other means, there is no passed-object dummy argument, so an explicit actual argument must be provided in the reference (as in '`call my_obp_sub(32, a)`' in Figure 14.2).

Figure 14.2 Associating the invoking object with the dummy argument `x`.

```
type t
   procedure(obp), pointer, pass(x) :: p
end type
abstract interface
   subroutine obp(w, x)
      import   :: t
      integer  :: w
      class(t) :: x
   end subroutine
end interface
⋮
type(t) a
a%p => my_obp_sub
⋮
call a%p(32)   ! equivalent to 'call my_obp_sub(32, a)'
```

The `pass` attribute may also be used to confirm the default (of passing the invoking object to the first dummy argument) by using the name of the first dummy argument.

If it is not desired to pass the invoking object to the procedure at all, the `nopass` attribute is used.

14.2.4 Internal procedures as targets of a procedure pointer

An internal procedure can be used as the target of a procedure pointer. When it is invoked via the corresponding procedure pointer, it has access to the variables of the host procedure as if it had been invoked there. When the host procedure returns, the procedure pointer will become undefined because the environment necessary for the evaluation of the internal procedure will have disappeared. For example, in Figure 14.3, on return from `sub` the variable `n` no longer exists for `f` to refer to.

Figure 14.3 Unsafe pointer to internal procedure.

```
module unsafe
  procedure(real), pointer :: funptr
contains
  subroutine sub(n)
    funptr => f    ! Associates funptr with internal function f.
    call process   ! funptr will remain associated with f during the
                   ! execution of subroutine "process".
    return         ! Returning from sub makes funptr become undefined.
  contains
    real function f(x)
      real, intent(in) :: x
      f = x**n
    end function
  end subroutine
end module
```

Exercises

1. Design a derived type to specify user-defined error handling; it should have a place for returning an error code and error message, and a procedure pointer component to allow for an 'error callback'.
2. Write an event queue (data structure) and event dispatcher (procedure) using procedure pointer components. Each event should have a time and an action (procedure to be invoked); the action procedures should take the time as an argument. There should be a `schedule` procedure which, given a time and a procedure, queues an event for that time. If the time has already passed, the procedure should still be enqueued for immediate activation. The dispatcher procedure itself should, on invocation, process each event in the queue in time order (including extra events scheduled during this process) until the queue is empty.

15. Object-oriented programming

15.1 Introduction

The object-oriented approach to programming and design is characterized by its focus on the data structures of a program rather than the procedures. Often, invoking a procedure with a data object as its principal argument is thought of as 'sending a message' to the object. Typically, special language support is available for collecting these procedures (sometimes known as 'methods') together with the definition of the type of the object.

This approach is supported in Fortran by type extension, polymorphic variables, and type-bound procedures.

15.2 Type extension

Type extension creates new derived types by extending existing derived types. To create a new type extending an old one, the `extends` attribute is used on the type definition statement. For example, given an old type such as

```
type person
   character(len=10) :: name
   real              :: age
   integer           :: id
end type person
```

this can be extended to form a new type with

```
type, extends(person) :: employee
   integer :: national_insurance_number
   real    :: salary
end type employee
```

The old type is known as the **parent type**. The new type inherits all the components of the parent type by a process known as **inheritace association** and may have additional components. So an `employee` variable has the inherited components of `name`, `age`, and `id`, and the additional components of `national_insurance_number` and `salary`. Where the order matters, that is, in a structure constructor that does not use keywords[1] and in default

[1] The use of keywords in structure constructors was described in Section 8.19.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

derived type input/output (Section 10.3), the inherited components come first in their order, followed by the new components in their order.

Additionally, an extended type has a **parent component**; this is a component that has the type and type parameters of the old type and its name is that of the old type. It allows the inherited portion to be referenced as a whole. Thus, an `employee` variable has a component called `person` of type `person`, associated with the inherited components. For example, given

```
type(employee) :: director
```

the component `director%name` is the same as `director%person%name`, and so on. The parent component is particularly useful when invoking procedures that operate on the parent type but which were not written with type extension in mind. For example, the procedure

```
subroutine display_older_people(parray, min_age)
   type(person), intent(in) :: parray(:)
   integer, intent(in)      :: min_age
   intrinsic                :: size
   do i=1, size(parray)
      if (parray(i)%age >= min_age) print *, parray(i)%name
   end do
end subroutine display_older_people
```

may be used with an array of `type(employee)` by passing it the parent component of the array, for example

```
type(employee) :: staff_list(:)
⋮
! Show the employees eligible for early retirement
call display_older_people(staff_list%person, 55)
```

The parent component is not ordered with respect to the other components, so no value for it can appear in a structure constructor unless keywords are used (see Section 8.19).

The parent component is itself inherited if the type is further extended (becoming a 'grandparent component'); for example, with

```
type, extends(employee) :: salesman
   real :: commission_rate
end type salesman
type(salesman) :: traveller
```

the `traveller` has both the `employee` and `person` components, and `traveller%person` is exactly the same as `traveller%employee%person`.

A type can be extended without adding components, for example

```
type, extends(employee) :: clerical_staff_member
end type clerical_staff_member
```

Although a `clerical_staff_member` has the same ultimate components as an `employee`, it is nonetheless considered to be a different type.

Extending a type without adding components can be useful in several situations, in particular:

- to create a type with additional operations (as specific or generic type-bound procedures, see Section 15.8.2);
- to create a type with different effects for existing operations, by overriding (Section 15.8.3) specific type-bound procedures; and
- for classification, that is, when the only extra information about the new type is the fact that it is of that type (for example, as in the `clerical_staff_member` type above).

A derived type is extensible (can be extended) provided it does not have the `sequence` (Appendix A.2) or `bind` (Section 19.4) attribute, or is `c_ptr` or `c_funptr` (Section 19.3). An extended type must not be given the `sequence` or `bind` attribute.

15.2.1 Type extension and type parameters

When a type is extended, the new type inherits all of the type parameters. New type parameters may also be added, for example:

```
type matrix(real_kind, n, m)
   integer, kind   :: real_kind
   integer, len    :: n, m
   real(real_kind) :: value(n, m)
end type matrix
type, extends(matrix) :: labelled_matrix(max_label_length)
   integer, len                :: max_label_length
   character(max_label_length) :: label = ''
end type labelled_matrix
type(labelled_matrix(kind(0.0), 10, 20, 200)) :: x
```

The variable `x` has four type parameters: `real_kind`, `n`, `m`, and `max_label_length`.

15.3 Polymorphic entities

15.3.1 Introduction to polymorphic entities

A polymorphic variable is a variable whose data type may vary at run time. It must be a pointer or allocatable variable, or a dummy data object, and is declared using the `class` keyword in place of the `type` keyword. For example,

```
type point
   real :: x, y
end type point
class(point), pointer :: p
```

declares a pointer `p` that may point to any object whose type is in the class of types consisting of `type(point)` and all of its extensions.

We say that the polymorphic object is **type compatible** with any object of the same declared type or any of its extensions.[2] A polymorphic pointer may only be pointer-associated

[2] A non-polymorphic object is type compatible only with objects of the same declared type.

with a type-compatible target, a polymorphic allocatable variable may only be allocated to have a type-compatible allocation (see Section 15.4.3), and a polymorphic dummy argument may only be argument-associated with a type-compatible actual argument. Furthermore, if a polymorphic dummy argument is allocatable or a pointer, the actual argument must be of the same declared type; this is to ensure that the type-compatibility relationship is enforced.

The type named in the `class` attribute must be an extensible derived type – it cannot be a `sequence` derived type, a `bind` derived type, or an intrinsic type. This type is called the **declared type** of the polymorphic entity, and the type of the object to which it refers is called the **dynamic type**.

However, even when a polymorphic entity is referring to an object of an extended type, it provides access via component notation only to components, type parameters, and bindings (see Section 15.8) of the declared type. This is because the compiler only knows about the declared type of the object, it cannot know about the dynamic type (which may vary at run time). Access to components, etc. that are in the dynamic type but not the declared type is provided by the `select type` construct (see Section 15.7).

A polymorphic dummy argument that is neither allocatable nor a pointer assumes its dynamic type from the actual argument. This provides a convenient means of writing a function that applies to any extension of a type, for example

```
real function distance(a, b)
  class(point) :: a, b
  distance = sqrt((a%x-b%x)**2 + (a%y-b%y)**2)
end function distance
```

This function will work unchanged, for example, not only on a scalar of type `point` but also on a scalar of type

```
type, extends(point) :: data_point
   real, allocatable :: data_value(:)
end type data_point
```

15.3.2 Establishing the dynamic type

A polymorphic dummy variable has its dynamic type and type parameters established by argument association and they do not vary during a single execution of the procedure, though they may be different on different invocations. However, the dynamic type and type parameters of a polymorphic allocatable or pointer variable can be altered at any time, as follows:

- it can be allocated to be of a type and type parameters specified on the `allocate` statement, see Section 15.4;
- using the `source=` specifier on the `allocate` statement, it can be allocated to have the same type, type parameters, and value as another variable;

the dynamic type of a polymorphic allocatable variable can be altered:

- when an allocation is transferred from one allocatable variable to another using the intrinsic subroutine `move_alloc` (see Section 6.8), the receiving variable takes on the dynamic type, type parameters, and value that the sender had;

and the dynamic type of a polymorphic pointer variable can be altered:

- via pointer association since a polymorphic pointer has the dynamic type of its target.

Note that an `allocate` statement that lacks both a type specification and the `source=` specifier will allocate the variable to be of its declared type.

The dynamic type and type parameters of a disassociated pointer or unallocated allocatable variable is its declared type. A pointer with undefined association status has no defined dynamic type: it is not permitted to be used in any context where its dynamic type would be relevant.

The dynamic type of an allocatable variable can also change due to automatic reallocation in an intrinsic assignment statement, see Section 15.5.

15.3.3 Limitations on the use of a polymorphic variable

A polymorphic variable may appear in an input/output list only if it is processed by defined derived-type input/output (Section 11.6).

The variable in an intrinsic assignment statement is not permitted to be polymorphic unless it is allocatable. However, if it is associated with a non-polymorphic variable, perhaps via the `type is` guard in a `select type` statement (see Section 15.7), assigning to the non-polymorphic variable will have the desired effect.

In a pure procedure, a polymorphic variable is not permitted to be an intent `out` dummy argument and no statement that might result in the deallocation of a polymorphic entity is permitted. A polymorphic variable is not permitted to be an actual argument corresponding to an intent `out` assumed-size dummy argument (see Section 19.5).

15.3.4 Polymorphic arrays and scalars

A polymorphic variable can be either an array or a scalar (including an allocatable scalar).

A polymorphic array is always homogeneous; that is, each array element has the same dynamic type and type parameters. This is by construction: every method for establishing the dynamic type of a polymorphic variable provides a single type for the entire array. The reason for this is both to make reasoning about progams simpler and to ensure that accessing an element of a polymorphic array is reasonably efficient.

If a heterogeneous polymorphic array is required, the usual trick of using an array of derived type with a scalar polymorphic pointer or allocatable component is available.

15.3.5 Unlimited polymorphic entities

Sometimes one wishes to have a pointer that may refer not just to objects in a class of extended types, but to objects of any type, perhaps even including non-extensible or intrinsic types. For example, one might wish to have a 'universal' list of variables (pointer targets), each of which might be of any type.

This can be done with an **unlimited polymorphic** pointer. These are declared using `*` as the `class` specifier, for example

```
class(*), pointer :: up
```

declares `up` to be an unlimited polymorphic pointer. This could be associated with a `real` target, for instance:

```
real, target :: x
⋮
up => x
```

The value of an unlimited polymorphic object cannot be accessed directly, but the object as a whole can be used; for example, it can be copied with `source=`, passed as an argument, used in pointer assignment, and most importantly it can be the selector in a `select type` statement (see Section 15.7).

Type information is maintained for an unlimited polymorphic pointer while it is associated with an intrinsic type or an extensible derived type, but not when it is associated with a non-extensible derived type. (This is because different non-extensible types are considered to be the same if they have the same structure and names.) To prevent a pointer of intrinsic or extensible type from becoming associated with an incompatible target, such a pointer is not permitted to be the left-hand side of a pointer assignment if the target is unlimited polymorphic. For example,

```
use iso_c_binding
type, bind(c) :: triplet
   real(c_double) :: values(3)
end type triplet
class(*), pointer      :: univp
type(triplet), pointer :: tripp
real, pointer          :: realp
⋮
univp => tripp           ! Valid
univp => realp           ! Valid
⋮
tripp => univp           ! Valid when the dynamic type matches
realp => univp           ! Always invalid
```

Instead of the invalid pointer assignment, a `select type` construct must be used to associate a pointer of intrinsic or extensible type with an unlimited polymorphic target. A longer example showing the use of unlimited polymorphic pointers, together with `select type`, is shown in Figure 15.4.

When an unlimited polymorphic pointer is allocated, the required type and type parameter values must be specified in the `allocate` statement (Section 15.4).

15.3.6 Polymorphic entities and generic resolution

Because a polymorphic dummy argument may be associated with an actual argument of an extended type, a polymorphic dummy argument is not distinguishable from a dummy argument of an extended type in the rules for distinguishing procedures in a generic set (Section 5.18). For example, the procedure

```
real function data_distance(a, b)
  class(data_point) :: a, b
  data_distance = ...
end function data_distance
```

is not permitted in the same generic set as the function `distance` defined in Section 15.3.1. Where such an effect is required, type-bound procedures (Section 15.8.3) may be employed.

In the case of an unlimited polymorphic dummy argument, because it is type-compatible with any type, it is indistinguishable from any argument of the same rank (except as noted in Section 5.18).

15.4 Typed and sourced allocation

15.4.1 Introduction

As well as determining array size, the `allocate` statement can determine type parameter values, type (for a polymorphic variable), and value. The general form of the `allocate` statement (Section 6.5) is

`allocate (` *[type-spec* `::` *] allocation-list [,* *alloc-spec]* ... `)`

If *type-spec* is present, it takes the form of the type name followed by the type parameter values in parentheses, if any, for both intrinsic and derived types. Alternatively, an *alloc-spec* may be `source=`*expr* or `mold=`*expr*, where *expr* is an expression with which the *allocation* is type-compatible (see Section 15.3.1). If each *allocation* is for an array of the same rank, *expr* may be an array of that rank, otherwise *expr* must be scalar.

An `allocate` statement with a *type-spec* is **typed allocation**, and an `allocate` statement with a `source=` or `mold=` clause is **sourced allocation**. Only one of these clauses is allowed, so an `allocate` statement cannot both be a typed allocation and a sourced allocation. We now explain these features.

15.4.2 Typed allocation and deferred type parameters

A length type parameter that is deferred (indicated by a colon in the *type-spec*) has no defined value until it is given one by an `allocate` statement or by pointer assignment (a type parameter that is not deferred cannot be altered by `allocate` or pointer assignment). For example, in

```
character(:), allocatable :: x(:)
⋮
allocate (character(n) :: x(m))
```

the array `x` will have `m` elements and each element will have character length `n` after execution of the `allocate` statement.

If a length parameter of an item being allocated is assumed, it must be specified as an asterisk in the *type-spec*. For example, the type parameter `string_dim` in Figure 15.1 must be specified as `*` because it is assumed.

Figure 15.1 Allocating a dummy argument with an assumed type parameter.

```
type string_vector(string_dim, space_dim)
   integer, len             :: string_dim, space_dim
   type(string(string_dim)) :: value(space_dim)
end type string_vector
⋮
subroutine allocate_string_vectors(vp, n, m)
   type(string_vector(*,:)), pointer :: vp(:)
   integer, intent(in)               :: n, m
   allocate (string_vector(string_dim=*, space_dim=n) :: vp(m))
end subroutine allocate_string_vectors
```

Note that there is only one *type-spec* in an `allocate` statement, so it must be suitable for all the items being allocated. In particular, if any one of them is a dummy argument with an assumed type parameter, they must all be dummy arguments that assume this type parameter.

If any type parameter is neither assumed nor deferred, the value specified for it by the *type-spec* must be the same as its current value. For example, in

```
subroutine allocate_string3_vectors(vp, n, m)
   type(string_vector(3,:)), pointer :: vp(:)
   integer, intent(in)               :: n, m
   allocate (string_vector(string_dim=3, space_dim=n) :: vp(m))
end subroutine allocate_string3_vectors
```

the expression provided for the `string_dim` type parameter must be equal to 3.

15.4.3 Polymorphic variables and typed allocation

For polymorphic variables, the *type-spec* specifies not only the values of any deferred type parameters, but also the dynamic type to allocate. If an item is unlimited polymorphic, it can be allocated to be any type (including intrinsic types); otherwise the type specified in the `allocate` statement must be an extension of the declared type of the item.

For example,

```
class(*), pointer :: ux, uy(:)
class(t), pointer :: x, y(:)
⋮
allocate (t2 :: ux, x, y(10))
allocate (real :: uy(100))
```

allocates `ux`, `x`, and `y` to be of type `t2` (an extension of `t`), and `uy` to be of type default `real`.

15.4.4 Sourced allocation

Instead of allocating a variable with an explicitly specified type (and type parameters), it is possible to take the type, type parameters, and value from another variable or expression.

This effectively produces a 'clone' of the source expression, and is done by using the `source=` clause in the `allocate` statement. For example, in

```
subroutine s(b)
  class(t), allocatable :: a
  class(t)              :: b
  allocate (a, source=b)
```

the variable `a` is allocated with the same dynamic type and type parameters as `b`, and will have the same value.

This is useful for copying heterogeneous data structures such as lists and trees, as in the example in Figure 15.2.

Figure 15.2 Allocating an object with the type and type parameters of another object.

```
type singly_linked_list
   class(singly_linked_list), pointer :: next => null()
   ! No data - the user of the type should extend it to include
   ! desired data.
end type singly_linked_list
⋮
recursive function sll_copy(source) result(copy)
   class(singly_linked_list), pointer    :: copy
   class(singly_linked_list), intent(in) :: source
   allocate (copy, source=source)
   if (associated(source%next)) copy%next => sll_copy(source%next)
end function sll_copy
```

If the allocated item is an array, its bounds and shape are specified in the usual way and are not taken from the source. This allows the source to be a scalar whose value is given to every element of the array. Alternatively, it may be an array of the same shape.

As we have seen, making a clone of an array can be done as follows:

```
class(t), allocatable :: a(:), b(:)
⋮
allocate (a(lbound(b,1):ubound(b,1)), source=b)
```

However, the bounds may be omitted, in which case they will be taken from the `source=` specifier, allowing the much simpler

```
allocate (b, source=a)
```

Further, there is a facility to allocate a variable to the shape, type, and type parameters of an expression without copying its value. This is done with the `mold=` specifier, for example

```
allocate (b, mold=a)
```

After the allocation any relevant default initialization will be applied to `b`.

Finally, both `mold=` and `source=` may be used when allocating multiple objects, for example,

```
allocate (a(10), b(20), source=173)
```

15.5 Assignment for allocatable polymorphic variables

Intrinsic assignment to an allocatable polymorphic variable is allowed, and this extends the automatic reallocation feature permitting array shape and deferred type parameters to handle types.

If the variable is allocated and its dynamic type and type parameters differ from that of the expression, the variable is deallocated (just as if it were an array with different shape or had different deferred type parameter values). If the variable was unallocated, or is deallocated by the previous step, it is allocated to have the dynamic type of the expression (and array bounds or type parameter values, if applicable). Finally, the value is copied just as in normal assignment (with shallow copying for any pointer components and deep copying for any allocatable components).

An example is

```
class(*), allocatable :: x
⋮
x = 3
```

The effect of automatic reallocation is similar to that of

```
if (allocated(variable)) deallocate (variable)
allocate (variable, source=expression)
```

except that, in the intrinsic assignment case,

- the variable may appear in the expression, and any reallocation occurs after evaluation of the expression and before the copying of the value; and
- if the variable is already allocated with the correct type (and shape and deferred type parameter values, if applicable), no reallocation is done; apart from performance, this only matters when the variable also has the `target` attribute and there is a pointer associated with it: instead of the pointer becoming undefined, it will remain associated and will see the new value.

15.6 The associate construct

The `associate` construct allows one to associate a name either with a variable or with the value of an expression, for the duration of a block. Any entity with this name outside the construct is separate and inaccessible inside it. The association is known as **construct association**. During execution of the block, the *associate-name* remains associated with the variable (or retains the value) specified, and takes its type, type parameters, and rank from its association. This construct is useful for simplifying multiple accesses to a variable which has a lengthy description (subscripts and component names). For example, given a nested set of derived-type definitions, the innermost of which is

```
type one
   real, allocatable, dimension(:) :: xvec, levels
   logical                         :: tracing
end type one
```

then the association as specified in

```
associate(point_qfstate => master_list%item(n)%qfield%posn(i, j)%state)
   point_qfstate%xvec = matmul(transpose_matrix, point_qfstate%xvec)
   point_qfstate%levels = timestep(point_qfstate%levels, input_field)
   if (point_qfstate%tracing) call show_qfstate(point_qfstate, stepno)
end associate
```

would be even less comprehensible if `point_qfstate` were written out in full in each occurrence.

Formally, the syntax is

[name `:` *]* `associate (` *association-list* `)`
block
`end associate` *[name]*

where each *association* is

associate-name `=>` *selector*

and *selector* is either a variable or an expression. As with other constructs, the `associate` construct can be named; if *name*`:` appears on the `associate` statement, the same name must appear on the `end associate` statement.

If the association is with a variable that does not have a vector subscript, the *associate-name* may be used as a variable within the block. The association is as for argument association of a dummy argument that does not have the `pointer` or `allocatable` attribute, but the *associate-name* has the `target` attribute if the variable does. If the association is with an expression, the *associate-name* may be used only for its value. If the association is with an array, the bounds of *associate-name* are given by the intrinsic functions `lbound` and `ubound` applied to the array.

If the *selector* is polymorphic, *associate-name* is also polymorphic. If *selector* is a pointer or has the `target` attribute, *associate-name* has the `target` attribute. The only other attributes that *associate-name* receives from the *selector* are the `asynchronous` and `volatile` attributes; in particular, if *selector* has the `optional` attribute, *associate-name* does not and so *selector* must be present when the construct is executed.

Multiple associations may be established within a single `associate` construct. For example, in

```
associate ( x => arg(i)%ground%coordinates(1), &
            y => arg(i)%ground%coordinates(2) )
   distance = sqrt((myloc%x-x)**2+(myloc%y-y)**2)
   bearing = atan2(myloc%y-y, myloc%x-x)
end associate
```

the simplifying names `x` and `y` improve the readability of the code.

Without this construct, to make this kind of code readable either a procedure would need to be used, or pointers (requiring, in addition, the `target` attribute on the affected variables). This could adversely affect the performance of the program (and indeed would probably still not attain the readability shown here).

The construct may be nested with other constructs in the usual way.

15.7 The select type construct

To execute alternative code depending on the dynamic type and type parameters of a polymorphic entity and to gain access to the dynamic parts, the `select type` construct is provided. If the entity is not unlimited polymorphic, this construct takes the form

```
[ name: ] select type ( [ associate-name =>] selector)
[ type-guard-stmt [ name ]
    block ] ...
end select [ name ]
```

where each type guard statement is one of

```
type is (derived-type-spec)
type is (intrinsic-type [ (type-parameter-value-list) ] )
class is (derived-type-spec)
class default
```

where *derived-type-spec* is defined in Section 13.2.1. A type guard that specifies an intrinsic type is only permitted if the *selector* is unlimited polymorphic. The *derived-type-spec* is required to be an extensible type that is type-compatible with the *selector*. As with other constructs, the `select type` construct can be named; if *name*: appears on the `select type` statement, the same name must appear on the `end select` statement and may appear on a type guard statement.

The selector is a variable or an expression and the *associate-name* is associated with it within the block by **construct association** in exactly the same way as for an associate construct (Section 15.6). However, the body is now divided into parts, at most one of which is executed as follows:

i) The block following a `type is` guard is executed if the dynamic type of the selector is exactly the derived type specified, and the kind type parameter values match.

ii) Failing this, the block following a `class is` guard is executed if it is the only one for which the dynamic type is the derived type specified, or an extension thereof, and the kind type parameter values match. If there is more than one such guard, one of them must be of a type that is an extension of the types of all the others, and its block is executed.

iii) Failing this, the block following a `class default` guard is executed.

In the (frequently occurring) case where the *selector* is a simple name and the same name is suitable for the *associate-name*, the '*associate-name*=>' may be omitted.

The example in Figure 15.3 shows a typical use of `select type`. Each type guard statement that specifies an extended type provides access via component notation to the extended components. Note that within a `type is` block the *associate-name* is not polymorphic, since it is known that its dynamic type is precisely the same as the type declared in the `type is` statement.

If the *derived-type-spec* contains a *type-param-spec-list*, values corresponding to kind type parameters must be constant expressions and those for length type parameters must be

Figure 15.3 Using the `select type` construct for polymorphic objects of class `particle`.

```
subroutine describe_particle(p)
   class(particle) :: p

! These attributes are common to all particles.
   call describe_vector('Position:',p%position)
   call describe_vector('Velocity:',p%velocity)
   print *,'Mass:',p%mass
! Check for other attributes.
   select type (p)
   type is (charged_particle)
      print *,'Charge:',p%charge
   class is (charged_particle)
      print *,'Charge:',p%charge
      print *,'... may have other (unknown) attributes.'
   type is (particle)
      ! Just the basic particle type, there is nothing extra.
   class default
      print *,'... may have other (unknown) attributes.'
   end select
end subroutine describe_particle
```

asterisks. This is so that length type parameters do not participate in type parameter matching, but are always assumed from the *selector*.

If the selector is unlimited polymorphic, a type guard statement is permitted to specify an intrinsic type, but still cannot specify a `sequence` or `bind` derived type. For example, if the unlimited polymorphic pointer `up` is associated with the `real` target `x`, the execution of

```
select type(up)
type is (real)
   up = 3.5
   rp => up
end select
```

assigns the value of 3.5 to `x` and associates the `real` pointer `rp` with `x`. (The pointer assignment would not have been allowed outside of the `select type` construct.)

A longer example, showing the use of unlimited polymorphic in constructing a generic vector list package, is shown in Figure 15.4.

15.8 Type-bound procedures

In object-oriented programming one often wishes to invoke a procedure to perform a task whose nature varies according to the dynamic type and type parameters of a polymorphic object.

Figure 15.4 Generic vector list and type selection.

```
type generic_vector_pointer_list_elt
   class(*), pointer                 :: element_vector(:) => null()
   procedure(gvp_processor), pointer :: default_processor => null()
   type(generic_vector_pointer_list_elt),pointer :: next => null()
end type generic_vector_pointer_list_elt
abstract interface
   subroutine gvp_processor(gvp)
      import :: generic_vector_pointer_list_elt
      class(generic_vector_pointer_list_elt) :: gvp
   end subroutine gvp_processor
end interface
type(generic_vector_pointer_list_elt), pointer :: p
⋮
do
   if (.not.associated(p)) exit
   select type(q => p%element_vector)
   type is (integer(selected_int_kind(9)))
      call special_process_i9(q)
   type is (real)
      call special_process_default_real(q)
   type is (double precision)
      call special_process_double_precision(q)
   type is (character(*))
      call special_process_character(q)
   class default
      if (associated(p%default_processor)) call p%default_processor
   end select
   p => p%next
end do
```

This is the purpose of **type-bound procedures**. These are procedures which are invoked through an object, and the actual procedure executed depends on the dynamic type and type parameters of the object. This is sometimes known as dynamic dispatch.

They are called type-bound because the selection of the procedure depends on the type and type parameters of the object, in contrast to procedure pointer components which depend on the value of the object (one might call the latter object-bound).

In some other languages type-bound procedures are known as methods, and invocation of a method is thought of as 'sending a message' to the object.

However, type-bound procedures can be used even when there is no intention to extend the type. We will first describe how to define and use type-bound procedures in the simple case, and later explain how they are affected by type extension.

15.8.1 Specific type-bound procedures

The type-bound procedure section of a derived-type definition is separated from the component section by the `contains` statement, analogous to the way that module variables are separated from the module procedures. The default accessibility of type-bound procedures is separate from the default accessibility for components; that is, even with `private` components, each type-bound procedure is `public` unless a `private` statement appears in the type-bound procedure section or unless it is explicitly declared to be `private`.

Figure 15.5 A type with two type-bound procedures.

```
module mytype_module
   type mytype
      private
      real :: myvalue(4) = 0.0
   contains
      procedure :: write => write_mytype
      procedure :: reset
   end type mytype
   private :: write_mytype, reset
contains
   subroutine write_mytype(this, unit)
      class(mytype)     :: this
      integer, optional :: unit
      if (present(unit)) then
         write (unit, *) this%myvalue
      else
         print *,this%myvalue
      end if
   end subroutine write_mytype
   subroutine reset(variable)
      class(mytype) :: variable
      variable%myvalue = 0.0
   end subroutine reset
end module mytype_module
```

Each type-bound procedure declaration specifies the name of the **binding**, and the name of the actual procedure to which it is bound. (The latter may be omitted if it is the same as the type-bound procedure name.) For example, in Figure 15.5 objects of type `mytype` have two type-bound procedures, `write` and `reset`. These are invoked as if they were component procedure pointers of the object, and the invoking object is normally passed to the procedure as its first argument. For example, the procedure references

```
call x%write(6)
call x%reset
```

are equivalent to

```
call write_mytype(x,6)
call reset(x)
```

However, because they are public, the type-bound procedures (`write` and `reset`) can be referenced anywhere in the program that has a `type(mytype)` variable, whereas, because the module procedures (`write_mytype` and `reset`) are private, they can only be directly referenced from within `mytype_module`.

The full syntax of the statement declaring specific type-bound procedures is

`procedure (`*interface-name*`),` *binding-attr-list* `::` *binding-name-list*

or

`procedure` *[[,* *binding-attr-list* *]* `::` *] type-bound-proc-decl-list*

where each *binding-attr* is one of

`public` or `private`
`deferred`
`non_overridable`
`nopass` or `pass` *[* `(`*arg-name*`)` *]*

each *type-bound-proc-decl* is *tbp-name [* `=>` *proc-name]* and each *interface-name* or *proc-name* is the name of a procedure with an explicit interface. The `public` and `private` attributes are permitted only in the specification part of a module. The `pass` and `nopass` attributes are described in Section 14.2.3. The form with *interface-name* is for declaring deferred bindings, so it must contain the `deferred` attribute; these are described in Section 15.10. The form without *interface-name* declares ordinary type-bound procedures, so it must not contain the `deferred` attribute. An example of the case where it is not desired to pass the invoking object is shown in Figure 15.6.

A type-bound procedure declaration statement may take a list of procedure bindings, so that multiple type-bound procedures can be declared in a single statement, as in

```
type mycomplex
   ⋮
contains
   procedure :: i_plus_myc, myc_plus_i, myc_plus_myc=>myc_plus, &
                myc_plus_r, r_plus_myc
   ⋮
end type
```

This can be a significant improvement when a type has many type-bound procedures.

If the `non_overridable` attribute appears, that type-bound procedure cannot be overridden during type extension (see Section 15.8.3). Note that `non_overridable` is incompatible with `deferred`, since that requires the type-bound procedure to be overridden.

15.8.2 Generic type-bound procedures

Type-bound procedures may be generic. A generic type-bound procedure is defined with the `generic` statement within the type-bound procedure part. This statement takes the form

Figure 15.6 Two type-bound procedures with the `nopass` attribute.

```
module utility_module
   private
   type, public :: utility_access_type
   contains
      procedure, nopass :: startup
      procedure, nopass :: shutdown
   end type
contains
   subroutine startup
      print *,'Process started'
   end subroutine
   subroutine shutdown
      stop 'Process stopped'
   end subroutine
end module
⋮
use utility_module
type(utility_access_type) :: process_control
call process_control%startup
```

`generic` *[[, access-spec] ::*] *generic-spec* `=>` *tbp-name-list*

and can be used for named generics as well as for operators, assignment, and user-defined derived-type input/output specifications. Each *tbp-name* specifies an individual (specific) type-bound procedure to be included in the generic set. As always, procedures with the same *generic-spec* must all be subroutines or all be functions.

For example, in Figure 15.7 the type-bound procedure `extract` is generic, being resolved to one of the specific type-bound procedures `xi` or `xc`, depending on the data type of the argument. Thus, in

```
use container_module
type(container) v
integer ix
complex cx
⋮
call v%extract(ix)
call v%extract(cx)
```

one of the '`extract_`*something*`_from_container`' procedures will be invoked.

A generic type-bound procedure need not be named; it may be an operator, assignment, or a user-defined derived-type input/output specification. In this case, the object through which the type-bound procedure is invoked is whichever of the operands corresponds to the passed-object dummy argument. For this reason, the specific type-bound procedures for an unnamed generic must not have the `nopass` attribute. Like other type-bound procedures,

Figure 15.7 A named generic type-bound procedure.

```
module container_module
   private
   type, public :: container
      integer, private :: i = 0
      complex, private :: c = (0.,0.)
   contains
      private
      procedure :: xi => extract_integer_from_container
      procedure :: xc => extract_complex_from_container
      generic, public :: extract => xi, xc
   end type
contains
   subroutine extract_integer_from_container(this, val)
      class(container), intent(in) :: this
      integer, intent(out)         :: val
      val = this%i
   end subroutine extract_integer_from_container
   subroutine extract_complex_from_container(this, val)
      class(container), intent(in) :: this
      complex, intent(out)         :: val
      val = this%c
   end subroutine extract_complex_from_container
end module container_module
```

unnamed generics that are public are accessible wherever the type or an object of the type is accessible.[3]

This is useful for packaging-up a type and its operations, because the `only` clause of a `use` statement does not affect the accessibility of type-bound operators, unlike operators defined by an interface block. This prevents the accidental omission of required operators by making a mistake in the `use` statement. This is particularly germane when using defined assignment between objects of the same type, since omitting the defined assignment would cause an unwanted intrinsic assignment to be used without warning.

For example, Figure 15.8 shows the overloading of the operator `(+)` for operations on `type(mycomplex)`; these operations are available even if the user has done

```
use mycomplex_module, only: mycomplex
```

15.8.3 Type extension and type-bound procedures

When a type is extended, the new type usually inherits all the type-bound procedures of the old type, as is illustrated in Figure 15.9, where the new type `charged_particle` inherits not only the components of `particle`, but also its type-bound procedures `momentum` and `energy`.

[3] See, for Fortran 2018, an extension to this statement, described in Section 23.4.1.

Figure 15.8 A generic type-bound operator.

```
module mycomplex_module
   type mycomplex
      private
      ⋮ ! data components not shown
   contains
      procedure          :: mycomplex_plus_mycomplex
      procedure          :: mycomplex_plus_real
      procedure, pass(b) :: real_plus_mycomplex
      generic, public :: operator(+) => mycomplex_plus_mycomplex, &
                         mycomplex_plus_real, real_plus_mycomplex
      procedure          :: conjg => mycomplex_conjg
      ⋮ ! many other operations and functions...
   end type
contains
   ⋮ ! procedures which implement the operations
end module
```

Specific type-bound procedures defined by the new type are either additional bindings (with a new name), or may **override** type-bound procedures that would otherwise have been inherited from the old type. (However, overriding a type-bound procedure is not permitted if the inherited one has the `non_overridable` attribute.) An overriding type-bound procedure binding must have exactly the same interface as the overridden procedure except for the type of the passed-object dummy argument; if there is a passed-object dummy argument, the overriding procedure must specify its type to be `class(`*new-type*`)`. Generic type-bound procedures defined by the new type always extend the generic set; the complete set of generic bindings for any particular generic identifier (including both the inherited and newly defined

Figure 15.9 Extending a type with type-bound procedures.

```
type particle
   type(vector) :: position, velocity
   real         :: mass
contains
   procedure :: momentum => particle_momentum
   procedure :: energy => particle_energy
end type particle

type, extends(particle) :: charged_particle
   real :: charge
end type charged_particle
```

generic bindings) must satisfy the usual rules for generic disambiguation (Sections 5.18 and 15.3.6). A procedure that would be part of an inherited generic set may be overridden using its specific name.

For example, in Figure 15.10 the three specific type-bound procedures have been overridden; when the generic operation of (+) is applied to entities of type `instrumented_mycomplex`, one of the overriding procedures will be invoked.

Figure 15.10 Extending the type of Figure 15.8 with overriding of type-bound procedures.

```
type, extends(mycomplex) :: instrumented_mycomplex
   integer, public :: plus_op_count = 0
contains
   procedure :: mycomplex_plus_mycomplex => instrumented_myc_plus_myc
   procedure :: mycomplex_plus_real => instrumented_myc_plus_r
   procedure :: real_plus_mycomplex => instr_r_p_myc
end type instrumented_mycomplex
```

15.9 Design for overriding

When designing a type that can be overridden, there are several important points that should be considered.

Firstly, the specific procedures in a generic set need to be `public`, otherwise they cannot be overridden. This is why `mycomplex_plus_mycomplex` in Figure 15.8 is `public`; not because the user would want to invoke it directly, but to name the operation so it can be overridden.

Secondly, because the declared type of a function result cannot be changed when overriding, if a function has a result of the type, and when extended would be expected to return the extended type, the function result should be polymorphic and allocatable. For example, the result of `mycomplex_conjg` should be `class(mycomplex), allocatable`; in `instrumented_mycomplex`, the overriding function can then allocate its result to have a dynamic type of `instrumented_mycomplex`.

Thirdly, if a function implements a binary operation, because dynamic dispatch only occurs via one of the arguments, it needs to check whether the other argument is further extended, and if so, manually dispatch via that argument. For a commutative operation like addition, this is simply a matter of swapping the operands and invoking the type-bound addition again.

An example of `mycomplex_plus_mycomplex` that illustrates these points is in Figure 15.11, and an example of `instrumented_myc_plus_myc` with this design is in Figure 15.12.

A fourth consideration is that if an operation is not commutative, providing for multiple dispatch will require the 'reverse' function also to be provided. For example, in the case of subtraction (`a − b`), because `b − a` would give the wrong result, we would need a function that computes '`b .rsub. a`'.

Figure 15.11 Providing for multiple dispatch.

```
recursive function mycomplex_plus_mycomplex(a, b) result(r)
  class(mycomplex), intent(in) :: a, b
  class(mycomplex), allocatable :: r
  if (.not.same_type_as(a, b) .and. extends_type_of(a, b)) then
    ! Dispatch via b in case it has overridden this procedure.
    r = b + a
  else
    allocate(r)
    .
    . ! Do the addition and assign to the components of r.
  end if
end function
```

Figure 15.12 Overriding and multiple dispatch.

```
function instrumented_myc_plus_myc (a,b) result(r)
 class(instrumented_mycomplex), intent(in) :: a
 class(mycomplex), intent(in) :: b
 class(mycomplex), allocatable :: r
 if (.not.same_type_as(a,b) .and. extends_type_of(a,b)) then
   ! Dispatch via b in case it has overridden this procedure.
   r = b + a
 else
   r = a
   select type(r)
   class is (instrumented_mycomplex) ! Always true.
     r%plus_op_count = a%plus_op_count + 1
     select type(b)
     class is (instrumented_mycomplex)
       r%plus_oper_count = r%plus_op_count + b%plus_op_count
       ! Use the mycomplex plus operation to do the actual addition.
       r%mycomplex = r%mycomplex + b%mycomplex
     class default
       ! Use the mycomplex plus operation to do the actual addition.
       r%mycomplex = r%mycomplex + b
     end select
   end select
 end if
end function
```

Finally, although the implied extra memory allocation can easily be eliminated by the compiler for `r = b + a`, this is not the case for the additions to `r%mycomplex`. These could be avoided if `mycomplex` also provided an 'add to' function, that adds a `mycomplex` value to a `mycomplex` variable; in this case the actual addition invocations in Figure 15.12 would be replaced by `call r%mycomplex%addto(...)`, where '`...`' is `b%mycomplex` for the first actual addition call, and `b` for the second. (Although making the components public so that `instrumented_myc_plus_myc` could do the addition manually itself would also avoid the extra memory allocation overhead, that would break the encapsulation of `mycomplex`, making it difficult if not impossible to later change its internal representation.)

15.10 Deferred bindings and abstract types

Sometimes a type is defined not for the purpose of creating objects of that type, but only to serve as a base type for extension. In this situation, a type-bound procedure in the base type might have no default or natural implementation, but rather only a well-defined purpose and interface. This is supported by the `abstract` keyword on the type definition and the `deferred` keyword in the `procedure` statement.

A simple example is shown in Figure 15.13. Here, the intention is that extensions of the type would have components that hold data about the file and `my_open` would be overridden by a procedure that uses these data to open it.

Figure 15.13 An abstract type.

```
type, abstract :: file_handle
contains
   procedure (open_file), deferred, pass :: my_open
   ⋮
end type file_handle
abstract interface
   subroutine open_file(handle)
      import                              :: file_handle
      class(file_handle), intent(inout) :: handle
   end subroutine open_file
end interface
```

The procedure is known as a **deferred** type-bound procedure. An interface is required, which may be an abstract interface or that of a procedure with an explicit interface.

No ordinary variable is permitted to be of an abstract type, but a polymorphic variable may have it as its declared type. When an abstract type is extended, the new type may be a normal extended type or may itself be abstract. Deferred bindings are allowed only in abstract types. (But an abstract type is not required to have any deferred binding.)

Figure 15.14 shows the definition of an abstract type `my_numeric_type`, and the creation of the normal type `my_integer_type` as an extension of it. Variables that are declared to be `my_numeric_type` must be polymorphic, and if they are pointer or allocatable the `allocate` statement must specify a normal type (see Section 15.4).

Figure 15.14 Abstract numeric type.

```
type, abstract :: my_numeric_type
contains
   private
   procedure(op2), deferred :: add
   procedure(op2), deferred :: subtract
   ⋮ ! procedures for other operations not shown
   generic, public :: operator(+) => add, ...
   generic, public :: operator(-) => subtract, ...
   ⋮ ! generic specs for other operations not shown
end type my_numeric_type
abstract interface
   function op2(a, b) result(r)
      import :: my_numeric_type
      class(my_numeric_type), intent(in)  :: a, b
      class(my_numeric_type), allocatable :: r
   end function op2
end interface
type, extends(my_numeric_type) :: my_integer
   integer, private :: value
contains
   procedure :: add => add_my_integer
   procedure :: subtract => subtract_my_integer
   ⋮
end type my_integer
```

The use of the `abstract` and `deferred` attributes ensures that objects of insufficient type cannot be created, and that when extending the abstract type to create a normal type, the programmer can expect a diagnostic from the compiler if he or she has forgotten to override any inherited deferred type-bound procedures.

15.11 Finalization

When variables are deallocated or otherwise cease to exist, it is sometimes desirable to execute some procedure which 'cleans up' after the variable, perhaps releasing some resource (such as closing a file or deallocating a pointer component). This process is known as **finalization** and is provided by **final subroutines**. Finalization is only available for derived types that do not have the `sequence` attribute (Appendix A.2) or the `bind` attribute (Section 19.4). Array constructors and structure constructors are not finalized.

The set of final subroutines for a derived type is specified by statements of the form

`final` *[* `::` *] subroutine-name-list*

in the type-bound procedure section; however, they are not type-bound procedures, and have no name which can be accessed through an object of the type. Instead, they execute automatically when an object of that type ceases to exist.

A final subroutine for a type must be a module procedure with a single dummy argument of that type. All the final subroutines for that type form a generic set and must satisfy the rules for unambiguous generic references; since they each have exactly one dummy argument of the same type, this simply means that the dummy arguments must have different kind type parameter values or rank. Each such dummy argument must be a variable without the `allocatable`, `intent(out)`, `optional`, `pointer`, or `value` attribute, and any length type parameter must be assumed (the value must be '`*`').

A non-pointer object is **finalizable** if its type has a final subroutine whose dummy argument matches the object. When a finalizable object is about to cease to exist (for example, by being deallocated or the execution of a `return` statement), the final subroutine is invoked with the object as its actual argument. This also occurs (in the called procedure) when the object is passed to an intent `out` dummy argument, or is the variable on the left-hand side of an intrinsic assignment statement. In the latter case, the final subroutine is invoked after the expression on the right-hand side has been evaluated, but before it is assigned to the variable.

An example is shown in Figure 15.15. When subroutine `s` returns, the subroutine `close_scalar_file_handle` will be invoked with `x` as its actual argument, and `close_rank1_file_handle` will be invoked with `y` as its actual argument. The order in which these will be invoked is processor dependent.

Figure 15.15 An example of finalization.

```
module file_handle_module
   type file_handle
      private
      ⋮
   contains
      final :: close_scalar_file_handle, close_rank1_file_handle
   end type file_handle
contains
   subroutine close_scalar_file_handle(h)
      type(file_handle) :: h
      ⋮
   end subroutine close_scalar_file_handle
   ⋮
end module file_handle_module
⋮
subroutine s(n)
   type(file_handle) :: x, y(n)
   ⋮
end subroutine s
```

Termination of a program by an error condition, or by execution of a `stop` statement or the `end` statement in the main program, does not invoke any final subroutines.

If an object contains any (non-pointer) finalizable components, the object as a whole will be finalized before the individual components. That is, in Figure 15.16, when `ovalue` is finalized, `destroy_outer_ftype` will be invoked with `ovalue` as its argument before `destroy_inner_ftype` is invoked with `ovalue%ivalue` as its argument.

Figure 15.16 A finalizable type with a finalizable component.

```
type inner_ftype
   ⋮
contains
   final :: destroy_inner_ftype
end type inner_ftype
type outer_ftype
   type(inner_ftype) :: ivalue
contains
   final :: destroy_outer_ftype
end type outer_ftype
⋮
type(outer_ftype) :: ovalue
```

15.11.1 Type extension and final subroutines

When a type is extended, the new type does not inherit any of the final subroutines of the old type. The new type is, however, still finalizable, and when it is finalized any applicable final subroutines of the old type are invoked on the parent component.

If the new type defines any final subroutine, it will be invoked before any final subroutines of the old type are invoked. (Which is to say, the object as a whole is finalized, then its parent component is finalized, etc.) This operates recursively, so that when `x` is deallocated in the code of Figure 15.17, `destroy_bottom_type` will be invoked with `x` as its argument, then `destroy_top_type` will be invoked with `x%top_type` as its argument.

15.12 Procedure encapsulation example

A procedure may require its user to define the problem to be solved by providing a function as well as data. The example that we will consider here is that of multi-dimensional quadrature, where the function to be integrated must be specified. This function may depend on other data in some complicated way that was not anticipated by the writer of the quadrature procedure.

Previously available solutions for problems of this kind have been:

i) for the quadrature procedure to accept an extra argument, typically a `real` vector, and pass that to the user-defined function when it is called;

Figure 15.17 Nested extensions of finalizable types.

```
type top_type
   ⋮
contains
   final :: destroy_top_type
end type
type, extends(top_type) :: middle_type
   ⋮
end type
type, extends(middle_type) :: bottom_type
   ⋮
contains
   final :: destroy_bottom_type
end type

type(bottom_type), pointer :: x
allocate (x)
⋮
deallocate (x)
```

ii) for the program to pass the information to the function via `module` variables or `common` blocks; or

iii) the use of 'reverse communication' techniques, where the program repeatedly calls the quadrature procedure giving it extra information each time, until the quadrature procedure is satisfied.

These all have disadvantages; the first is not very flexible (a `real` vector might be a poor way of representing the data), the second requires global data (recognized as being poor practice) and is not thread-safe, while the third is flexible and thread-safe but very complicated to use, particularly for the writer of the quadrature procedure.

With type extension, the user can package up a procedure with any kind of required data, and the quadrature procedure will pass the data through. Figure 15.18 shows the definition of the types concerned and an outline of the quadrature procedure. Details not relevant to the function evaluation (such as the definition of the types for passing options to the procedure, and for receiving the status of the integration) have been omitted.

To use `ndim_integral`, the user needs to extend the abstract type to include any necessary data components and to bind his or her function to the type. Figure 15.19 shows how the user could do this for an arbitrary polynomial function.

To actually perform an integration, the user merely needs a local variable of this type to be loaded with the required data, and calls the quadrature procedure as shown in Figure 15.20.

Figure 15.18 Outline of a quadrature module.

```
module quadrature_module
   integer, parameter :: wp = selected_real_kind(15)
   type, abstract :: bound_user_function
      ! No data components
   contains
      procedure(user_function_interface), deferred :: eval
   end type bound_user_function
   abstract interface
      real(wp) function user_function_interface(data, coords)
         import                       :: wp, bound_user_function
         class(bound_user_function) :: data
         real(wp), intent(in)       :: coords(:)
      end function user_function_interface
   end interface
   ⋮
contains
   real(wp) function ndim_integral(hyper_rect, userfun, options, &
                                   status)
      real(wp), intent(in)                  :: hyper_rect(:)
      class(bound_user_function)            :: userfun
      type(quadrature_options), intent(in) :: options
      type(quadrature_status), intent(out) :: status
      ⋮
      ! This is how the user function is invoked
      single_value = userfun%eval(coordinates)
      ⋮
   end function ndim_integral
   ⋮
end module
```

Figure 15.19 Extending the Figure 15.18 type for polynomial integration.

```
module polynomial_integration
   use quadrature_module

   type, extends(bound_user_function) :: my_bound_polynomial
      integer              :: degree, dimensionality
      real(wp),allocatable :: coeffs(:,:)
   contains
      procedure :: eval => polynomial_evaluation
   end type
contains
   real(wp) function polynomial_evaluation(data, coords) result(r)
      class(my_bound_polynomial) :: data
      real(wp), intent(in)       :: coords(:)
      integer                    :: i, j
      r = 0
      do i=1, data%dimensionality
        r = r + sum([ (data%coeffs(i, j)*coords(i)**j, &
                                          j=1, data%degree) ])
      end do
   end function polynomial_evaluation
end module polynomial_integration
```

Figure 15.20 Performing polynomial integration.

```
use polynomial_integration
type(my_bound_polynomial) :: poly
real(wp)                  :: integral
real(wp), allocatable     :: hyper_rectangle(:)
type(quadrature_options)  :: options
type(quadrature_status)   :: status

! Read the data into the local variable
read (...) poly%degree, poly%dimensionality
allocate (poly%coeffs(poly%dimensionality, poly%degree))
read (...) poly%coeffs

! Read the hyper-rectangle information
allocate (hyper_rectangle(poly%dimensionality))
read (...) hyper_rectangle
⋮
! Option-setting omitted
! Evaluate the integral
integral = ndim_integral(hyper_rectangle, poly, options, status)
```

15.13 Type inquiry functions

There are two intrinsic functions for comparing dynamic types. These are intended for use on polymorphic variables, but may also be used on non-polymorphic variables.

`extends_type_of(a, mold)` returns, as a scalar default logical, whether the dynamic type of `a` is an extension of the dynamic type of `mold`. Both `a` and `mold` must either be unlimited polymorphic or of extensible type.

This will return true if `mold` is unlimited polymorphic and is either a disassociated pointer or an unallocated allocatable variable; otherwise, if `a` is unlimited polymorphic and is either a disassociated pointer or an unallocated allocatable variable, it will return false.

Otherwise, if both `a` and `mold` are unlimited polymorphic and neither has extensible dynamic type, the result is processor dependent.

`same_type_as(a, b)` returns, as a scalar default logical, whether the dynamic type of `a` is the same as the dynamic type of `b`. Both `a` and `b` must either be unlimited polymorphic or of extensible type.

If both `a` and `b` are unlimited polymorphic and neither has extensible dynamic type, the result is processor dependent.

For both functions, neither argument is permitted to be a pointer with undefined association status.[4]

These two functions are not terribly useful, because knowing the dynamic type of `a` (or how it relates to the dynamic type of `b` or `mold`) does not in itself allow access to the extended components. Therefore, we recommend that `select type` be used for testing the dynamic types of polymorphic entities.

Exercises

1. Define a polygon type where each point is defined by a component of class `point` (defined in Section 15.3). A function to test whether a position is within the polygon would be useful. A typical extension of such a type could have a label and some associated data; define such an extension.
2. Define a data logging type. This should contain type-bound procedures to initialize logging to a particular file, and to write a log entry. The file should automatically be closed if the object ceases to exist.
3. Use pointer functions to implement a vector that counts how many times it is accessed as a whole vector and how many times a single element from it is accessed.

[4] But, for Fortran 2018, see Section 23.7.2.

16. Submodules

16.1 Introduction

The module facilities that we have described so far are those of Fortran 95. While adequate for programs of modest size, they have some shortcomings for very large programs. These shortcomings all arise from the fact that, although modules are an aid to modularization of the program, they are themselves difficult to modularize. As a module grows larger, perhaps because the concept it is encapsulating is large, the only modularization mechanism is to break it into several modules. This exposes the internal structure, raising the potential for unnecessary global name clashes and giving the user of the module access to what ought to be private data and/or procedures. Worse, if the subfeatures of the module are interconnected, they must remain together in a single module, however large.

Another significant shortcoming is that if a change is made to the code inside a module procedure, even a private one, typical use of `make` or similar tools results in the recompilation of every file which `use`d that module, directly or indirectly.

The solution is to allow modules to be split into separate program units called submodules, which can be in separate files. Module procedures can then be split so that the interface information remains in the module, but the bodies can be placed in the submodules. A change in a submodule cannot alter an interface, and so does not cause the recompilation of program units that use the module. A submodule has access via host association to entities in the module, and may have entities of its own in addition to providing implementations of module procedures.

Submodules give other benefits, which we can explain more easily once we have described the feature.

16.2 Separate module procedures

The essence of the feature is to separate the definition of a module procedure into two parts: the interface, which is defined in the module; and the body, which is defined in the submodule. Such a module procedure is known as a **separate module procedure**. A simple example is shown in Figure 16.1. The keyword `module` in the prefix of the `function` statement indicates in the interface block that this is the interface to a module procedure rather than an external procedure, and in the submodule that this is the implementation part of a module procedure. The `submodule` specifies the name of its parent. Both the interface and the submodule gain access to the type `point` by host association.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

Figure 16.1 A separate module procedure.

```
module points
   type :: point
      real :: x, y
   end type point
   interface
      real module function point_dist(a, b)
         type(point), intent(in) :: a, b
      end function point_dist
   end interface
end module points

submodule (points) points_a
contains
   real module function point_dist(a, b)
      type(point), intent(in) :: a, b
      point_dist = sqrt((a%x-b%x)**2+(a%y-b%y)**2)
   end function point_dist
end submodule points_a
```

The interface specified in the submodule must be exactly the same as that specified in the interface block. For an external procedure, the interface is permitted to differ, for example, in respect of the names of the arguments, whether it is pure, and whether it is recursive (see Section 5.11); such variations are not permitted for a submodule since the intention is simply to separate the definition of the procedure into two parts. The name of the result variable is not part of the interface and so is permitted to be different in the two places; in this case, the name in the interface block is ignored.

An alternative is to give none of the interface information is the submodule. Instead, the whole interface is taken from the interface block in the module, including whether it is a function or a subroutine and the name of the result variable if it is a function. Here is an example

```
submodule (points) points_a
contains
   module procedure point_dist
      point_dist = sqrt((a%x-b%x)**2+(a%y-b%y)**2)
   end procedure point_dist
end submodule points_a
```

Note the use of the keyword `procedure`, which avoids specifying whether it is a function or a subroutine.

16.3 Submodules of submodules

Submodules are themselves permitted to have submodules, which is useful for very large programs. The module or submodule of which a submodule is a direct subsidiary is called its **parent** and it is called a **child** of its parent. We do not expect the number of levels of submodules often to exceed two (that is, a module with submodules that themselves have submodules) but there is no limit and we refer to **ancestors** and **descendants** with the obvious meanings. Each module or submodule is the root of a tree whose other nodes are its descendants and have access to it by host association. No other submodules have such access, which is helpful for developing parts of large modules independently. Furthermore, there is no mechanism for accessing anything declared in a submodule from elsewhere – it is effectively private.

If a change is made to a submodule, only it and its descendants will need recompilation.

To indicate that a submodule has a submodule as its parent, the module name in the `submodule` statement is replaced by *module-name*`:`*parent-name*. For example,

```
submodule (points:points_a) points_b
```

declares that `points_b` has the submodule of Figure 16.1 as its parent. Note that this syntax allows two submodules to have the same name if they are descendants of different modules, which is useful when they are developed independently.

16.4 Submodule entities

A submodule can also contain entities of its own. These are not module entities and so are neither public nor private; they are, however, inaccessible outside of the defining submodule except to its descendants.

Typically, these will be variables, types, named constants, etc., for use in the implementation of some separate module procedure. As per the usual rules of host association, if any submodule entity has the same name as a module entity, the module entity is hidden.

A submodule can also contain procedures, which we will call **submodule procedures**. A submodule procedure is only accessible in the submodule and its descendants, and so can be invoked only there. To ensure this property holds for a submodule procedure with the `bind` attribute (see Section 19.9), such a procedure is not permitted to have a `bind` attribute with a `name=` specifier. This means that it has no binding label, so there is no additional mechanism for invoking the procedure.

Like a module procedure, a submodule procedure can also be separate; a separate submodule procedure has its interface declared in one submodule and its body in a descendant.

16.5 Submodules and use association

A submodule may access its ancestor module by use association, but it can access entities therein by host association so this is usually unnecessary. It may also access another module by use association. In particular, it is possible for a submodule of module `one` to access module `two` and a submodule of module `two` to access module `one`. A simple example is

where a procedure of module `one` calls a procedure of module `two` and a procedure of module `two` calls a procedure of module `one`. Because circular dependencies between modules are not permitted, without submodules this would require that `one` and `two` were the same module, or that a third module `three` be used (containing those parts which were mutually dependent).

16.6 The advantages of submodules

A major benefit of submodules is that if a change is made to one, only it and its descendants are affected. Thus, a large module may be divided into small submodule trees, improving modularity (and thus maintainability) and avoiding unnecessary recompilation cascades. We now summarize other benefits.

Entities declared in a submodule are private to that submodule and its descendants, which controls their name management and accidental use within a large module.

Separate concepts with circular dependencies can be separated into different submodules in the common case where it is just the implementations that reference each other (because circular dependencies are not permitted between modules, this was impossible before).

Where a large task has been implemented as a set of modules, it may be appropriate to replace this by a single module and a collection of submodules. Entities that were public only because they are needed by other modules of the set can become private to the module or to a submodule and its descendants.

Once the implementation details of a module have been separated into submodules, the text of the module itself can be published to provide authoritative documentation of the interface without exposing any trade secrets contained in the implementation.

On many systems, each source file produces a single object file that must be loaded in its entirety into the executable program. Breaking the module into several files will allow the loading of only those procedures that are actually invoked into a user program. This makes modules more attractive for building large libraries.

17. Coarrays

17.1 Introduction

The coarray programming model is designed to provide a simple syntactic extension to support parallel programming from the point of view of both **work distribution** and **data distribution**.

Firstly, consider work distribution. The coarray extension adopts the Single Program Multiple Data (SPMD) programming model. A single program is replicated a fixed number of times, each replication having its own set of data objects. Each replication of the program is called an **image**. The number of images could be the same as, or more than or less than, the number of physical processors. A particular implementation may permit the number of images to be chosen at compile time, at link time, or at execution time. Each image executes asynchronously, and the normal rules of Fortran apply within each image.[1] The execution sequence can differ from image to image as specified by the programmer who, with the help of a unique image index, determines the actual path using normal Fortran control constructs and explicit synchronizations. For code between synchronizations, the compiler is free to use almost all its normal optimization techniques as if only one image were present, but with access to the additional memory that the other images provide.

Secondly, consider data distribution. The coarray extension allows the programmer to express data distribution by specifying the relationship between memory images in a syntax very much like normal Fortran array syntax. Coarray objects have an important property: as well as having access to the local object, each image may access the corresponding object on any other image. For example, the statements

```
real, dimension(1000), codimension[*] :: x, y
real, codimension[*] :: z
```

declare three objects, each as a **coarray**; `x` and `y` are array coarrays and `z` is a scalar coarray. A coarray always has the same shape on each image. In this example, each image has two real array coarrays of size 1000 and a scalar coarray. If an image executes the statement:

```
x(:) = y(:)[q]
```

the coarray `y` on image `q` is copied into coarray `x` on the executing image.

Subscripts within parentheses follow the normal Fortran rules within one image. **Cosubscripts** within square brackets provide an equally convenient notation for accessing an object

[1] Although this is not required, we expect implementations to arrange for each image to execute the same executable file on identical hardware.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

on another image. Bounds for cosubscripts are given in square brackets in coarray declarations, except that the final cobound must be an asterisk. The asterisk indicates that the cobound is determined by the number of images, given that the coarray exists on them all. This allows the programmer to write code without knowing the number of images the code will eventually use.

The programmer uses coarray syntax only where it is needed. A reference to a coarray with no square brackets attached to it is a reference to the object in the memory of the executing image. Since it is desirable for most references to data objects in a parallel program to be local, coarray syntax should appear only in isolated parts of the source code. Coarray syntax acts as a visual flag to the programmer that communication between images will take place. It also acts as a flag to the compiler to generate code that avoids latency[2] whenever possible.

Because a coarray has the same shape on every image, and because allocations and deallocations of coarrays occur in synchrony across all images, coarrays may be implemented in such a way that each image can calculate the address of a coarray on another image. This is sometimes called **symmetric memory**. On a shared-memory machine, a coarray on an image and the corresponding coarrays on other images might be implemented as a sequence of objects with evenly spaced addresses. On a distributed-memory machine with one physical processor for each image, a coarray might be stored at the same address in each physical processor. If it is an array coarray, each image can calculate the address of an element on another image relative to the array start address on that other image.

Because coarrays are integrated into the language, remote references automatically gain the services of Fortran's basic data capabilities, including

- the type system;
- automatic conversions in assignments;
- information about structure layout; and
- object-oriented features, but with some restrictions.

17.2 Referencing images

Data objects on other images are referenced by cosubscripts enclosed in square brackets. Each valid set of cosubscripts maps to an **image index**, which is an integer between one and the number of images, in the same way as a valid set of array subscripts maps to a position in the array element order.

The number of images is returned by the intrinsic function `num_images` with no arguments. The intrinsic function `this_image` with no arguments returns the image index of the invoking image. The set of cosubscripts that corresponds to the invoking image for a coarray `z` are available as the rank-one array `this_image(z)`. The image index that corresponds to an array `sub` of valid cosubscripts for a coarray `z` is available as `image_index(z, sub)`.

For example, on image 5, `this_image()` has the value `5` and, for the array coarray declared as

```
real :: z(10, 20)[10, 0:9, 0:*]
```

[2]Delay while the image waits for data to be transferred to or from another image.

`this_image(z)` has the value `[5, 0, 0]`, whilst on image 213, `this_image(z)` has the value `[3, 1, 2]`. On any image, the value of `image_index(z, [5, 0, 0])` is `5` and the value of `image_index(z, [3, 1, 2])` is `213`.

17.3 The properties of coarrays

Each image has its own set of data objects, all of which may be accessed in the normal Fortran way. Some objects are declared with **codimensions** in square brackets, for example:

```
real, dimension(20), codimension[20,*] :: a ! An array coarray
real  :: c[*], d[*]                         ! Scalar coarrays
character :: b(20)[20,0:*]
integer :: ib(10)[*]
type(interval) :: s[20,*]
```

A coarray may be allocatable, see Section 17.6. Unless the coarray is allocatable, it must have **explicit coshape**, that is, be declared with integer expressions for cobounds except that the final upper cobound must be an asterisk. The total number of subscripts plus cosubscripts is limited to 15.

A coarray is not permitted to be a pointer because maintaining the property of symmetric memory would impose severe restrictions. Furthermore, because an object of type `c_ptr` or `c_funptr` (Section 19.3) has the essence of a pointer, a coarray is not permitted to be of either of these types. However, a coarray may be of a derived type with pointer or allocatable components, see Section 17.7.[3]

A subobject of a coarray is regarded as a coarray if and only if it has no cosubscripts, no vector subscripts, no allocatable component selection, and no pointer component selection. For example, `a(1)`, `a(2:10)`, and `s%lower` are coarrays if `a` and `s` are the coarrays declared at the start of this section. This definition means that passing a coarray subobject to a dummy coarray does not involve copy-in copy-out (which would be infeasible given that the coarray exists on all images). Also, it ensures that the subobject has the property of symmetric memory. The term **whole coarray** is used for the whole of an object that is declared as a coarray or the whole of a coarray component of a structure.

The **corank** of a whole coarray is determined by its declaration. Its **cobounds** are specified within square brackets in its declaration or allocation. Any subobject of a whole coarray that is a coarray has the corank, cobounds, and coextents of the whole coarray. The cosize of a coarray is always equal to the number of images. Even though the final upper bound is specified as an asterisk, a coarray has a final coextent and a final upper cobound, which depend on the number of images. The final upper cobound is the largest value that the final cobound can have in a valid reference (we discuss this further in Section 17.4). For example, when the number of images is 128, the coarray declared thus

```
real :: array(10,20)[10,-1:8,0:*]
```

has rank `2`, corank `3`, and shape `[10,20]`; its lower cobounds are `1, -1, 0` and its upper cobounds are `10, 8, 1`.

[3] It is also is permitted to have a component of type `c_ptr` or `c_funptr`, but these are nearly useless, see Appendix A.9.2.

A coarray is not permitted to be a named constant, because this would be useless. Each image would hold exactly the same value so there would be no reason to access its value on another image.

To ensure that each image initializes only its own data, cosubscripts are not permitted in `data` statements. For example:

```
real :: a(10)[*]
data a(1)    /0.0/ ! Permitted
data a(1)[2] /0.0/ ! Not permitted
```

17.4 Accessing coarrays

A coarray on another image may be addressed by using cosubscripts in square brackets following any subscripts in parentheses, for example:

```
a(5)[3,7] = ib(5)[3]
d[3] = c
a(:)[2,3] = c[1]
```

but not after component selection, for example:

```
s%lower[1,1] = 0.0 ! Not permitted for a coarray s.
```

It is anomalous that `s%lower` is a coarray but cannot be accessed across images with cosubscripts. The reason is to allow coarray components (see Section 17.8).

We call any object whose designator includes cosubscripts a **coindexed object**. Only one image may be referenced at a time, so each cosubscript must be a scalar integer expression (section cosubscripts, such as `d[1:n]`, are not permitted). Section subscripts must be used when the coindexed object has nonzero rank. For example, `a[2,3]` is not permitted as a shorthand for `a(:)[2,3]`. This is in order to make it clear that the rank is nonzero.

Any object reference without square brackets is always a reference to the object on the executing image. For example, in

```
real :: z(20)[20,*], zmax[*]
⋮
zmax = maxval(z)
```

the value of the largest element of the array coarray `z` on the executing image is placed in the scalar coarray `zmax` on the executing image.

For a reference with square brackets, the cosubscript list must map to a valid image index. For example, if there are 16 images and the coarray `z` is declared thus

```
real :: z(10)[5,*]
```

then a reference to `z(:)[1,4]` is valid because it refers to image `16`, but a reference to `z(:)[2,4]` is invalid because it refers to image `17`. Like array subscripts, it is the programmer's responsibility to ensure that cosubscripts are within cobounds and refer to a valid image.

Square brackets attached to objects alert the reader to probable communication between images. However, communication may also take place during the execution of a procedure reference, and this could be via a defined operation or defined assignment.

That an image is selected in square brackets has no bearing on whether the statement is executed on that image. For example, the statement

```
z[6] = 1
```

is executed by every image that encounters it. If code is to be executed selectively, the Fortran `if` or `case` construct is needed. An example is

```
if (this_image()==6) z = 1
```

A coindexed object is permitted in most contexts, such as intrinsic operations, intrinsic assignment, input/output lists, and as an actual argument corresponding to a non-coarray dummy argument.[4] On a distributed-memory machine, passing it as an actual argument is likely to cause a local copy of it to be made before execution of the procedure starts (unless it has intent `out`) and the result to be copied back on return (unless it has intent `in` or the `value` attribute). The rules for argument association have been carefully constructed so that such copying is always allowed.

Pointers are not allowed to have targets on remote images, because this would break the requirement for remote access to be obvious. Therefore, the target of a pointer is not permitted to be a coindexed object:

```
p => a(n)[p]   ! Not allowed (compile-time constraint)
```

A coindexed object is not permitted as the *selector* in an `associate` or `select type` statement because that would disguise a reference to a remote image (the associated name is without square brackets). However, a coarray is permitted as the selector, in which case the associated entity is also a coarray and its cobounds are those of the selector.

17.5 The sync all statement

Each image executes on its own without regard to the execution of other images except when it encounters a special kind of statement called an **image control statement**. The programmer inserts image control statements to ensure that, whenever one image alters the value of a coarray variable, no other image still wants the old value, and that whenever an image accesses the value of a coarray variable, it receives the wanted value – either the old value (before the update) or the new value (from the update). Because a coarray may be of a derived type with a pointer component (see Section 17.7), coarray syntax might be used to access a variable that has the `target` attribute but is not a coarray. It follows that image control statements are needed for these variables, too. In this section we describe the simplest of these image control statements.

The `sync all` statement provides a barrier where all images synchronize before executing further statements. All statements executed before the barrier on image P execute before any statement executes after the barrier on image Q. If the value of a variable is changed by an image before the barrier, the new value is available to all other images after the barrier. If a

[4] Polymorphic coindexed objects are much more restricted, see Section 17.10.

variable is referenced before the barrier and is changed by another image after the barrier, the old value is obtained.

Figure 17.1 Read on image 1 and broadcast to the others.

```
real :: z[*]
⋮
sync all
if (this_image()==1) then
   read (*, *) z
   do image = 2, num_images()
      z[image] = z
   end do
end if
sync all
⋮
```

Figure 17.1 shows a simple example of the use of `sync all`. Image 1 reads data and broadcasts it to other images. The first `sync all` ensures that image 1 does not interfere with any previous use of `z` by another image. The second `sync all` ensures that another image does not access `z` before the new value has been set by image 1.

The full form of the `sync all` statement is

`sync all` *`[(`* *[sync-stat-list]* *`)]`*

where *sync-stat* is `stat=`*stat* or `errmsg=`*erm*. If the `stat=` specifier is present, the integer variable *stat* is given the value zero after a successful execution or a positive value otherwise. If the `errmsg=` specifier is present and an error occurs, an explanatory message is assigned to the character variable *erm*.

Although usually the synchronization will be initiated by the same `sync all` statement on all images, this is not a requirement. The additional flexibility may be useful, for example, when different images are executing different code and need to exchange data.

All images are synchronized at program initiation as if by a `sync all` statement. This ensures that initialized coarrays will have their initial values on all images before any image commences executing its executable statements.

There is an implicit barrier whenever a coarray is allocated or deallocated, see Section 17.6. Other image control statements are described in Sections 17.13, and a complete list is found in Section 17.13.7.

17.6 Allocatable coarrays

A coarray may be allocatable; it must have **deferred coshape**, that is, be declared with colons for all codimensions. It is given normal cobounds by the allocate statement, for example,

```
real, allocatable :: a(:)[:], s[:, :]
⋮
allocate (a(10)[*], s[-1:34,0:*])
```

The cobounds must always be included in the `allocate` statement and the upper bound for the final codimension must always be an asterisk. For example, the following are not permitted (compile-time constraints):

```
allocate (a(n))      ! Not allowed for a coarray (no cobounds)
allocate (a(n)[p])   ! Not allowed (cobound not *)
```

Also, the value of each bound, cobound, or length type parameter is required to be the same on all images. For example, the following is not permitted (run-time constraint):

```
allocate (a(this_image())[*])  ! Not allowed (varying local bound)
```

Furthermore, if the dynamic types and type parameters are given on the `allocate` statement by typed or sourced allocation (Section 15.4), they must be the same on all images. Together, these restrictions ensure that the coarrays exist on every image and are consistent.

The `move_alloc` intrinsic subroutine may be applied to coarrays of the same corank.

There is implicit barrier synchronization of all images in association with each `allocate` statement that involves one or more coarrays. Images do not commence executing subsequent statements until all images finish executing the same `allocate` statement (on the same line of the source code). Similarly, for the `deallocate` and `move_alloc` statements for coarrays, all images synchronize at the beginning of the same statement and do not continue with the next statement until all images have finished executing the statement. The synchronization means that `move_alloc` is not pure when applied to coarrays.

When an image executes an `allocate` statement, communication is needed between images only for synchronization. The image allocates its local coarray and records how the corresponding coarrays on other images are to be addressed. The compiler is not required to check that the bounds and cobounds are the same on all images, although it may do so (or have an option to do so). Nor is the compiler required to detect when deadlock has occurred; for example, when one image is executing an `allocate` statement while another is executing a `deallocate` statement.

If an unsaved allocatable coarray is local to a procedure or block construct (Section 8.13), and is still allocated when the procedure or block construct completes execution, implicit deallocation of the coarray and therefore synchronization of all images occurs.

The allocation of a polymorphic coarray is not permitted to create a coarray that is of type `c_ptr`, `c_funptr`, or of a type with a coarray ultimate component.

Fortran allows the shapes or length parameters to disagree on the two sides of an intrinsic array assignment to an allocatable array (see Section 6.7); the system performs the appropriate reallocation. Such disagreement is not permitted for an allocatable coarray because it would imply synchronization.

For the same reason, intrinsic assignment is not permitted to a polymorphic coarray.

17.7 Coarrays with allocatable or pointer components

A coarray is permitted to be of a derived type with allocatable or pointer components.

17.7.1 Data components

To share data structures with different sizes, length parameter values, or types between different images, we may declare a coarray of a derived type with a non-coarray component that is allocatable or a pointer. On each image, the component is allocated locally or is pointer assigned to a local target, so that it has the desired properties on that image (or is not allocated or pointer assigned if it is not needed on that image). It is straightforward to access such data on another image, for example

```
x(:) = z[p]%alloc(:)
```

where the cosubscript is associated with the scalar variable `z`, not with its component. In words, this statement means 'Go to image `p`, obtain the address of the array component `alloc`, and copy the data in the array itself to the local array `x`'.

If coarray `z` contains a data pointer component `ptr`, the appearance of `z[q]%ptr` in a context that refers to its target is a reference to the target of component `ptr` of `z` on image `q`. This target must reside on image `q` and must have been established by an `allocate` statement executed on image `q` or a pointer assignment executed on image `q`, for example

```
z%ptr => r ! Local association
```

A local pointer may be associated with a target component on the local image,

```
r => z%ptr ! Local association
```

but may not be associated with a target component on another image,

```
r => z[q]%ptr ! Not allowed (compile-time constraint)
```

If an association with a target component on another image would otherwise be implied, the pointer component becomes undefined. For example, this happens when the following derived-type intrinsic assignments are executed on an image other than `q`:

```
z[q] = z ! The pointer component of z[q] may become undefined
z = z[q] ! The pointer component of z may become undefined
```

It can also happen in a procedure invocation on an image other than `q` if `z[q]` is an actual argument or `z[q]%ptr` is associated with a pointer dummy argument.

Similarly, for a coarray of a derived type that has a pointer or allocatable component, `allocate`, `deallocate`, or `move_alloc` for the component on another image is not allowed:

```
type(something), allocatable ::  t[:]
⋮
allocate (t[*])        ! Allowed
allocate (t%ptr(n))    ! Allowed
allocate (t[q]%ptr(n)) ! Not allowed (compile-time constraint)
```

However, the `nullify` statement may be applied to a coindexed pointer.

In an intrinsic assignment to a coindexed object that is allocatable, the shapes and length type parameters are required to agree; this prevents any possibility of a remote allocation. For the same reason, intrinsic assignment to a polymorphic coindexed object or a coindexed object with an allocatable ultimate component is not permitted. Furthermore, if an actual argument is a coindexed object with an allocatable ultimate component, the corresponding dummy argument must be allocatable, a pointer, or have the intent `in` or `value` attribute.

17.7.2 Procedure pointer components

A coarray is permitted to be of a type that has a procedure pointer component or a type-bound procedure. Invoking a procedure pointer component of a coindexed object, for example

```
call a[p]%proc(x) ! Not allowed
```

is not permitted since the remote procedure target might be meaningless on the executing image. However, invoking a type-bound procedure (Section 15.8) is allowed except for a polymorphic subobject of a coindexed object, for example

```
call a[p]%tbp1(x)      ! Allowed even if a is polymorphic.
call a[p]%comp%tbp2(x) ! Not allowed if comp is polymorphic.
```

This ensures that the type and hence the procedure is the same on all images.

17.8 Coarray components

A component may be a coarray, and if so must be allocatable. A variable or component of a type that has an ultimate coarray component cannot itself be a coarray and must be a non-pointer non-allocatable scalar.[5]

If an object with a coarray ultimate component is declared without the `save` attribute in a procedure and the coarray is still allocated on return, there is an implicit deallocation and associated synchronization. Similarly, if such an object is declared within a `block` construct and the coarray is still allocated when the block completes execution, there is an implicit deallocation and associated synchronization.

To avoid the possibility of implicit reallocation in an intrinsic assignment for a scalar of a derived type with a coarray component, no disagreement of allocation status or shape is permitted for the coarray component.

It is not permissible to add a coarray component by type extension unless the type already has one or more coarray components.

17.9 Coarrays in procedures

A dummy argument of a procedure is permitted to be a coarray. It may be a scalar, or an array that is explicit shape, assumed size (Section 19.5) , assumed shape, or allocatable, see Figure 17.2. Unless it is allocatable (Section 17.6), its coshape has to be explicit.

When the procedure is called, the corresponding actual argument must be a coarray. The association is with the coarray itself and not with a copy; the restrictions below ensure that copy-in copy-out is never needed. (Making a copy would require synchronization on entry and return to ensure that remote references within the procedure are not to a copy that does not exist yet or that no longer exists.) Furthermore, the interface is required to be explicit so that the compiler knows it is passing the coarray and not just the local variable. An example is shown in Figure 17.3.

[5] Were we to allow a coarray of a type with coarray components, we would be confronted with references such as `z[p]%x[q]`. A logical way to read such an expression would be: go to image `p` and find component `x` on image `q`. This is equivalent to `z[q]%x`.

Figure 17.2 Coarray dummy arguments.

```
subroutine subr(n, p, u, w, x, y, z, a)
   integer :: n, p
   real :: u[2, p/2, *]            ! Scalar
   real :: w(n)[p, *]              ! Explicit shape
   real :: y(:, :)[2, *]           ! Assumed shape
   real, allocatable :: z(:)[:, :] ! Allocatable array
   real, allocatable :: a[:]       ! Allocatable scalar
```

Figure 17.3 Calling a procedure with coarray dummy arguments.

```
   real, allocatable :: a(:)[:], b(:,:)[:]
   ⋮
   call sub(a(:), b(1,:))
   ⋮
contains
   subroutine sub(x, y)
      real :: x(:)[*], y(:)[*]
      ⋮
      end subroutine sub
```

The restrictions on coarray dummy arguments are:

- the actual argument must be a coarray (see Section 17.3 for the rules on whether a subobject is a coarray);
- if the dummy argument is an array other than an assumed-shape array without the `contiguous` attribute (see Section 7.18.1), the actual argument must be **simply contiguous** (satisfies conditions given Section 7.18.2, which ensure that the array is known at compile time to be contiguous); and
- it must not have the `value` attribute (this also applies to a non-coarray dummy argument that has a coarray ultimate component).

If a dummy argument is an allocatable coarray, the corresponding actual argument must be an allocatable coarray of the same rank and corank. Furthermore, its chain of argument associations, perhaps through many levels of procedure call, must terminate with the same actual coarray on every image. This allows the coarray to be allocated or deallocated in the procedure.

If a dummy argument is an allocatable coarray or has an ultimate component that is a coarray, it must not have intent `out`. This is because the implicit deallocation (Section 6.9) of the coarray would also require an implicit synchronization.

A procedure with a non-allocatable coarray dummy argument will often be called on all images at the same time with the same actual coarray, but this is not a requirement. For example, the images may be grouped into two sets and the images of one set may be calling

the procedure with one coarray while the images of the other set are calling the procedure with another coarray or are executing different code.

Automatic coarrays are not permitted. For example, the following is invalid:

```
subroutine solve3(n)
integer :: n
real :: work(n)[*]  ! Not permitted
```

Were automatic coarrays permitted, it would be necessary to require synchronization, both after memory is allocated on entry and before memory is deallocated on return. Furthermore, it would mean that the procedure would need to be called on all images concurrently.

A function result is not permitted to be a coarray or to have an ultimate component that is a coarray. This is because the function would need to allocate temporary coarrays for its results, which would involve synchronization of all images for every invocation within a statement.

Allocatable coarrays may be declared in a procedure.

The rules for resolving generic procedure references have not been extended to allow overloading of array and coarray versions because this would be ambiguous.

A pure or elemental procedure is not permitted to define a coindexed object or contain any image control statements (Section 17.13), since these involve side-effects (defining a coindexed object is similar to defining a variable from the host or a module). However, it may reference the value of a coindexed object.

Neither a final subroutine (Section 15.11) nor an elemental procedure is permitted to have a coarray dummy argument.

Unless it is allocatable or a dummy argument, an object that is a coarray or has a coarray ultimate component is required to have the `save` attribute.[6] Again this is because an unsaved non-allocatable coarray would come into existence on procedure invocation, which would require synchronization, and this is inappropriate because procedures are not invoked in lockstep on every image. An allocatable coarray is not required to have the `save` attribute because its initial state is unallocated; note also that a recursive procedure may need separate allocatable coarrays at more than one level of recursion.

Each image associates its non-allocatable coarray dummy argument with an actual coarray, perhaps through many levels of procedure call, and defines the corank and cobounds afresh. It uses these to interpret each reference to a coindexed object, taking no account of whether a remote image is executing the same procedure with the corresponding coarray.

17.10 References to polymorphic subobjects

We need the term **potential subobject component** for a component of a type at any level of component selection as long as no selection is through a pointer. For example, given the types in Figure 17.4, the potential subobject components of type `two` are `ordinary`, `alloc`, `point`, `ordinary%comp`, and `ordinary%alloc%comp`, but not `ordinary%point%comp`. So that the implementation does not need to query the dynamic type of an object on another image, no references are permitted to a polymorphic subobject of a coindexed object or to a coindexed object of a type that has a polymorphic allocatable potential subobject component.

[6] Note that variables declared in a module or submodule, as well as main-program variables, automatically have the `save` attribute.

Figure 17.4 Nested types.

```
type one
   integer :: comp
end type one
type two
   type(one)              :: ordinary
   type(one), allocatable :: alloc(:)
   type(one), pointer     :: point
end type two
```

17.11 Volatile and asynchronous attributes

If a dummy coarray is volatile, so too must the corresponding actual argument be, and vice versa. Without this restriction, the value of a non-volatile coarray might be altered via another image by means not specified by the program, that is, behave as volatile.

Similarly, agreement of the attribute is required when accessing a coarray or a variable with a coarray ultimate component by use association, host association, or in a `block` construct (see Section 8.13) from the scope containing it. Here, the restriction is simple; since the attribute is the same by default, it must not be respecified for an accessed coarray.

For the same reason, agreement of the `volatile` attribute is required for pointer association with any part of a coarray.

An asynchronous or volatile coindexed object is not permitted to be an actual argument that corresponds to an asynchronous or volatile dummy argument unless the dummy argument has the `value` attribute. This is because passing a coindexed object as an actual argument is likely to be done by copy-in copy-out.

17.12 Interoperability

Coarrays are not interoperable, since C does not have the concept of a data object like a coarray. Interoperability of coarrays with UPC[7] might be considered in the future.

17.13 Synchronization

We have encountered barrier synchronization in Sections 17.5 and 17.6. Here, we describe the **image control statements** that provide more selective synchronizations, and the concept of the execution segment that underpins the behaviour of programs that employ them.

17.13.1 Execution segments

On each image, the sequence of statements executed before the first execution of an image control statement or between the execution of two image control statements is known as

[7] Unified Parallel C, an extension of C which is similar to coarrays in Fortran.

a **segment**. The segment executed immediately before the execution of an image control statement includes the evaluation of all expressions within the statement.

For example, in Figure 17.1, each image executes a segment before executing the first `sync all` statement, executes a segment between executing the two `sync all` statements, and executes a segment after executing the second `sync all` statement.

On each image P, the statement execution order determines the segment order, P_i, $i = 1, 2, \ldots$. Between images, the execution of corresponding image control statements on images P and Q at the end of segments P_i and Q_j may ensure that either P_i precedes Q_{j+1}, or Q_j precedes P_{i+1}, or both.

A consequence is that the set of all segments on all images is partially ordered: the segment P_i precedes segment Q_j if and only if there is a sequence of segments starting with P_i and ending with Q_j such that each segment of the sequence precedes the next either because they are on the same image or because of the execution of corresponding image control statements.

A pair of segments P_i and Q_j are called **unordered** if P_i neither precedes nor succeeds Q_j. For example, if the middle segment of Figure 17.1 is P_i on image 1 and Q_j on another image Q, P_{i-1} precedes Q_{j+1} and P_{i+1} succeeds Q_{j-1}, but P_i and Q_j are unordered.

There are restrictions on what is permitted in a segment that is unordered with respect to another segment. These provide the compiler with scope for optimization. A coarray may be defined and referenced during the execution of unordered segments by calls to atomic subroutines (Section 17.13.5). Apart from this,

- if a variable is defined in a segment on an image, it must not be referenced, defined, or become undefined in a segment on another image unless the segments are ordered;
- if the allocation of an allocatable subobject of a coarray or the pointer association of a pointer subobject of a coarray is changed in a segment on an image, that subobject shall not be referenced or defined in a segment on another image unless the segments are ordered; and
- if a procedure invocation on image P is in execution in segments P_i, P_{i+1}, ..., P_k and defines a non-coarray dummy argument, the argument-associated entity shall not be referenced or defined on another image Q in a segment Q_j unless Q_j precedes P_i or succeeds P_k (because a copy of the actual argument may be passed to the procedure).

It follows that for code in a segment, the compiler is free to use almost all its normal optimization techniques as if only one image were present.

17.13.2 The sync images statement

For greater flexibility, the `sync images` statement

`sync images (` *image-set[, sync-stat-list]* `)`

performs a synchronization of the image that executes it with each of the other images in its image set. Here, *image-set* is either an integer expression indicating distinct image indices or an asterisk indicating all images. The expression must be scalar or of rank one. The optional *sync-stat-list* is as for `sync all` (Section 17.5).

Execution of a `sync images` statement on image P corresponds to the execution of a `sync images` statement on image Q if the number of times image P has executed a `sync images`

statement with Q in its image set is the same as the number of times image Q has executed a `sync images` statement with P in its image set. The segments that executed before the `sync images` statement on either image precede the segments that execute after the corresponding `sync images` statement on the other image. Figure 17.5 shows an example that imposes the fixed order 1, 2, ... on images.

Figure 17.5 Using `sync images` to impose an order on images.

```
me = this_image()
ne = num_images()
if (me==1) then
   p = 1
else
   sync images (me-1)
   p = p[me-1] + 1
end if
if (me<ne) sync images (me+1)
```

Execution of a `sync images(*)` statement is not equivalent to the execution of a `sync all` statement. A `sync all` statement causes all images to wait for each other, whereas `sync images` statements are not required to specify the same image set on all the images participating in the synchronization. In the example in Figure 17.6, image 1 will wait for each of the other images to reach the `sync images(1)` statement. The other images wait for image 1 to set up the data, but do not wait for each other.

Figure 17.6 Using `sync images` to make other images wait for image 1.

```
if (this_image() == 1) then
   ! Set up coarray data needed by all other images
   sync images (*)
else
   sync images (1)
   ! Use the data set up by image 1
end if
```

17.13.3 The lock and unlock statements

Locks provide a mechanism for controlling access to data that are referenced or defined by more than one image.

A lock is a scalar variable of the derived type `lock_type` that is defined in the intrinsic module `iso_fortran_env`. The type has private components that are not pointers and are not allocatable. It does not have the `bind` attribute or any type parameters, and is not a sequence type. All components have default initialization. A lock must be a coarray or a subobject of a coarray. It has one of two states: **locked** or **unlocked**. The unlocked state is represented by a

single value and this is the initial value. All other values are locked. The only way to change the value of a lock is by executing the `lock` or `unlock` statement. For example, if a lock is a dummy argument or a subobject of a dummy argument, the dummy argument must not have intent `out`. If a lock variable is locked, it can be unlocked only by the image that locked it.

Figure 17.7 Using `lock` and `unlock` to manage stacks.

```
module stack_manager
   use, intrinsic :: iso_fortran_env, only: lock_type
   type task
   ⋮
   end type
   type(lock_type), private :: stack_lock[*]
   type(task), private      :: stack(100)[*]
   integer, private         :: stack_size[*]
   type(task), parameter    :: null = task(...)
contains
   subroutine get_task(job)
   ! Get a task from my stack
      type(task), intent(out) :: job
      lock (stack_lock)
      if (stack_size>0) then
         job = stack(stack_size)
         stack_size = stack_size - 1
      else
         job = null
      end if
      unlock (stack_lock)
   end subroutine get_task
   subroutine put_task(job, image)
   ! Put a task on the stack of image
      type(task), intent(in) :: job
      integer, intent(in)    :: image
      lock (stack_lock[image])
         stack_size[image] = stack_size[image] + 1
         stack(stack_size[image])[image] = job
      unlock (stack_lock[image])
   end subroutine put_task
end module stack_manager
```

Figure 17.7 illustrates the use of `lock` and `unlock` statements to manage stacks. Each image has its own stack; any image can add a task to any stack. If a `lock` statement is executed for a lock variable that is locked by another image, the image waits for the lock to be unlocked by that image. The effect in this example is that `get_task` has to wait if another

image is adding a task to the stack, and `put_task` has to wait if `get_task` is getting a task from the stack or another image is executing `put_task` for the same stack.

There is a form of the `lock` statement that avoids a wait when the lock variable is locked:

```
logical :: success
lock (stack_lock, acquired_lock=success)
```

If the variable is unlocked, it is locked and the value of `success` is set to true; otherwise, `success` is set to false and there is no wait.

An error condition occurs for a `lock` statement if the lock variable is already locked by the executing image, and for an `unlock` statement if the lock variable is not already locked by the executing image. As for the `allocate` and `deallocate` statements, the `stat=` specifier is available to avoid this causing error termination. The values `stat_locked`, `stat_locked_other_image`, and `stat_unlocked` from the intrinsic module `iso_fortran_env` are given to the `stat=` variable in a `lock` statement for a variable locked by the image, in a `lock` statement for a variable locked by another image, or in an `unlock` statement for a variable that is unlocked.

Any particular lock variable is successively locked and unlocked by a sequence of `lock` and `unlock` statements, each of which separates two segments on the executing image. If execution of such an `unlock` statement P_u on image P is immediately followed in this sequence by execution of a `lock` statement Q_l on image Q, the segment that precedes the execution of P_u on image P precedes the segment that follows the execution of Q_l on image Q.

The full syntax of the `lock` statement is

`lock (` *lock-variable*`[,` *lock-stat-list*`]` `)`

where *lock-stat* is `acquired_lock=`*scalar-logical-variable* or *sync-stat*. The full syntax of the `unlock` statement is

`unlock (` *lock-variable*`[,` *sync-stat-list*`]` `)`

In both cases, *sync-stat* is as for `sync all` (Section 17.5).

For a sourced allocation of a coarray (using `source=` to take its value from another variable or expression), the source expression is not permitted to be of type `lock_type` or have a potential subobject component of that type because this would create a new lock that might be locked initially.

17.13.4 Critical sections

Exceptionally, it may be necessary to limit execution of a piece of code to one image at a time. Such code is called a **critical section**. There is a construct to delimit a critical section:

```
critical
⋮  ! code that is executed on one image at a time
end critical
```

No image control statement may be executed during the execution of a critical construct, that is, the code executed must be a single segment. Branching into or out of a critical construct is not permitted.

If image Q is the next to execute the construct after image P, the segment in the critical section on image P precedes the segment in the critical section on image Q.

As for other constructs, the critical construct may be named:

```
example: critical
⋮  ! code that is executed on one image at a time
end critical example
```

17.13.5 Atomic subroutines and the sync memory statement

An **atomic subroutine** is an intrinsic subroutine that acts on a scalar variable `atom` of type `integer(atomic_int_kind)` or `logical(atomic_logical_kind)`, whose kind value is defined in the intrinsic module `iso_fortran_env`. The variable `atom` must be a coarray or a coindexed object. Two executions of atomic subroutines with the same `atom` actual argument are permitted to occur in unordered segments; the effect is as if each action on the argument `atom` occurs instantaneously and does not overlap with the other. The programmer has no control over which occurs first and it may vary from run to run on the same computer and the same compiled code. If the order matters, the programmer must add image control statements, but there can be performance gains where the order does not matter.

The execution of a `sync memory` statement defines a boundary on an image between two segments, each of which can be ordered in some user-defined way with respect to segments on other images. One way to effect user-defined ordering between images is by employing the atomic subroutines `atomic_define` and `atomic_ref`. We see the construction of reliable and portable code in this way as very difficult – it is all too easy to introduce subtle bugs that manifest themselves only occasionally. We therefore do not recommend this practice and defer its description to Appendix A.9.1. Other atomic subroutines are included in Fortran 2018 and are described in Section 20.20.

17.13.6 The stat= and errmsg= specifiers in synchronization statements

All the synchronization statements, that is, `sync all`, `sync images`, `lock`, `unlock`, and `sync memory`, have optional `stat=` and `errmsg=` specifiers for variables that must not be coindexed. They have the same role for these statements as they do for `allocate` and `deallocate` (Sections 6.6 and 6.5). No specifier may have a value that depends on the value of another.

If any of these statements, including `allocate` and `deallocate`, encounter an image that has executed a `stop` or `end program` statement and have a `stat=` specifier, the `stat=` variable is given the value of the constant `stat_stopped_image` from the `iso_fortran_env` intrinsic module. For `allocate` and `deallocate`, the action occurs on the executing images. For the others, the effect of executing the statement is otherwise the same as that of executing the `sync memory` statement. Without a `stat=` specifier, the execution of such a statement initiates error termination (Section 17.14).

17.13.7 The image control statements

The full list of **image control statements** is

- `sync all`;
- `sync images`;
- `lock` or `unlock`;
- `sync memory`;
- `allocate`, `deallocate`, or `call move_alloc` involving a coarray;
- `critical` or `end critical`;
- `end` or `return` that involves an implicit deallocation of a coarray;
- a statement that completes the execution of a `block` (see Section 8.13) and results in an implicit deallocation of a coarray;
- `stop` or `end program`.

17.14 Program termination

In a successful computation, all images need to continue executing until they have all executed a `stop` or `end program` statement, or referenced a procedure defined by a C companion processor and it terminates. This is called **normal termination**. On the other hand, if one of them encounters an error condition that terminates its execution, the computation is flawed and it is desirable to stop the other images as soon as is practicable. This is called **error termination**.

Normal termination occurs in three steps: initiation, synchronization, and completion. An image initiates normal termination but its data need to be accessible to other images until they have all initiated termination. Hence there needs to be synchronization, after which all images can complete execution.

An image initiates error termination if it executes a statement that causes error termination. This causes all other images that have not already initiated error termination to initiate error termination. Within the performance limits of the processor's ability to send signals to other images, this is expected to terminate all images immediately.

The statement

`error stop` *[stop-code]*

where *stop-code* is an integer or default character constant expression, explicitly initiates error termination. When executed on one image, this will cause all other images that have not already initiated error termination to initiate error termination. It thus causes the whole calculation to stop as soon as is practicable. The meaning of *stop-code* is the same as for the `stop` statement, see Section 5.3.[8]

The example in Figure 17.8 illustrates the use of `stop` and `error stop` in a climate model that uses two sets of images, one for the ocean and one for the atmosphere.

[8]This statement is extended in Fortran 2018, see Section 23.5.

Figure 17.8 `stop` and `error stop` in a climate model.

```
program climate_model
   use, intrinsic :: iso_fortran_env, only: stat_stopped_image
   integer, allocatable :: ocean_set(:), atmosphere_set(:)
   integer :: i, sync_stat
   ⋮
! Form two sets
   ocean_set = [ (i,i=1,num_images()/2) ]
   atmosphere_set = [ (i,i=1+num_images()/2,num_images()) ]
   ⋮
! Perform independent calculations
   if (this_image() > num_images()/2) then
      call atmosphere(atmosphere_set)
   else
      call ocean(ocean_set)
   end if
! Wait for both sets to finish
   sync all (stat=sync_stat)
   if (sync_stat == stat_stopped_image) then
      ⋮ ! Preserve data on file
      stop
   end if
   call exchange_data ! Exchange data between sets
   ⋮
contains
   subroutine atmosphere (set)
      integer :: set(:)
      ⋮ ! Perform atmosphere calculation.
      if (...) then ! Something has gone slightly wrong
      ⋮ ! Preserve data on file
         stop
      end if
      ⋮
      if (...) error stop  ! Something has gone very badly wrong
      ⋮
      sync images (set, stat=sync_stat))
      if (sync_stat == stat_stopped_image) then
         ⋮ ! Remaining atmosphere images preserve data in a file
         stop
      end if
   end subroutine atmosphere
```

If something goes badly wrong in the atmosphere calculation, the whole model is invalid and a restart is impossible, so all images stop as soon as possible without trying to preserve any data.

If something goes slightly wrong with the atmosphere calculation, the images in the atmosphere set write their data to files and stop, but their data remain available to the ocean images which complete execution of the ocean subroutine. On return from the computation procedures, if something went slightly wrong with the atmosphere calculation, the ocean images write data to files and stop, ready for a restart in a later run.

17.15 Input/output

Just as each image has its own variables, so it has its own input/output units. Whenever an input/output statement uses an integer expression to index a unit, it refers to the unit on the executing image.

The default unit for input (`*` in a `read` statement or `input_unit` in the intrinsic module `iso_fortran_env`) is preconnected on image 1 only.

The default unit for output (`*` in a `write` statement or `output_unit` in the intrinsic module `iso_fortran_env`) and the unit that is identified by `error_unit` in the intrinsic module `iso_fortran_env` are preconnected on each image. The files to which these are connected are regarded as separate, but it is expected that the processor will merge their records into a single stream or a stream for all `output_unit` files and a stream for all `error_unit` files. If records from these separate files are being merged into one stream, the merging process is processor dependent. Synchronization and `flush` statements might be sufficient to control the ordering of records, but this is not guaranteed. For example, the processor might delay merging the files until termination.

Any other preconnected unit is connected on the executing image only, and the file is completely separate from any preconnected file on another image.

The `open` statement connects a file to a unit on the executing image only. Whether a file with a given name is the same file on all images or varies from one image to the next is processor dependent.

A file is not permitted to be connected to more than one image.[9]

17.16 Intrinsic procedures

The following intrinsic procedures are available. None are permitted in a constant expression. Again, we use italic square brackets *[ ]* to indicate optional arguments.

17.16.1 Inquiry functions

`image_index (coarray, sub)` returns a default integer scalar.

If `sub` holds a valid sequence of cosubscripts for `coarray`, the result is the corresponding image index. Otherwise, the result is zero.

[9] But see Section 23.15.5 where Fortran 2018 lifts this restriction.

coarray is a coarray that can have cosubscripts[10] and is of any type.
sub is a rank-one integer array of size equal to the corank of `coarray`.

lcobound (coarray[, dim][, kind]) returns the lower cobounds of a coarray in just the same way as `lbound` returns the lower bounds of an array.

ucobound (coarray[, dim][, kind]) returns the upper cobounds of a coarray in just the same way as `ubound` returns the upper bounds of an array.

17.16.2 Transformational functions

num_images () returns the number of images as a default integer scalar.

this_image () returns the index of the invoking image as a default integer scalar.

this_image (coarray) returns a default integer array of rank one and size equal to the corank of `coarray`; it holds the set of cosubscripts of `coarray` for data on the invoking image.
coarray is a coarray that can have cosubscripts[10] and is of any type.

this_image (coarray, dim) returns a default integer scalar holding cosubscript `dim` of `coarray` for data on the invoking image.
coarray is a coarray that can have cosubscripts[10] and is of any type.
dim is a default integer scalar whose value is in the range $1 \leq$ `dim` $\leq n$ where n is the corank of `coarray`.

17.17 Arrays of different sizes on different images

So far in this chapter we have dealt with the use of identically shaped coarrays distributed over all the images. By way of contrast, here we note that allocatable or pointer arrays that are not coarrays are allocated independently on different images without synchronization and thus may have differing shapes. Here is an example of code that sums all the elements of rank-one arrays on the images. The input arrays may have varying sizes.

```
real function sum_all(x)
  real, intent(in) :: x(:)
  real, save :: part_sum[*]
  sum_all = 0
  part_sum = sum(x)
  sync all
  if(this_image()==1)then
     do i = 1,num_images()
        sum_all = sum_all + part_sum[i]
     end do
  end if
end function sum_all
```

[10]The coarray `s%lower` of Section 17.4 is an example of a coarray that cannot have cosubscripts.

Exercises

1. Write a program in which image 1 reads a real value from a file and copies it to the other images, then all images print their values. Is a `sync all` statement needed before the printing?

2. Write a program in which there is an allocatable array coarray that is allocated of size 3, given values on all images by image 1, and then printed by all images. Is a `sync all` statement needed after the allocation?

3. Write a subroutine that has a scalar coarray argument and replaces it by the sum of all values across the images with only τ references to remote images, assuming that the number of images is 2^τ. Hint: Treat the images as in a circle and arrange that at the start of the ith loop, each image holds the sum of its original value and the next $2^i - 1$ original values.

4. Suppose we have a rectangular grid of size `nrow` by `ncol` with a real value at each point and `ncol==num_images()`. The first and last rows are regarded as neighbours and the the first and last columns are regarded as neighbours. If the values are distributed in the coarray `u(1:nrow)[*]`, write a subroutine with arguments `nrow`, `ncol`, and `u` that replaces each value by the sum of the values at its four neighbours minus four times its own value.

5. Suppose we have the coarrays `a(1:nx,1:ny)[*]` and `b(1:ny,1:nz)[*]`. Assuming that `max(nx,ny,nz)` $\leq$ `num_images()`, write code to copy the data in `b` to `a` with redistribution so that `a(i,j)[k] == b(j,k)[i]` for all valid values of the indices.

Does your code have any bottlenecks where the same image is being asked for data by many images? If so, modify it to avoid this.

6. Adapt your subroutine from Exercise 3 to apply to a set of images by adding an array argument holding the indices of the set and a scalar argument holding the position of the executing image in the set, assuming that the size of the set is a power of 2. In a main program, set up two sets and values in a coarray, then call your subroutine simultaneously for your two sets.

18. Floating-point exception handling

18.1 Introduction

Exception handling is required for the development of robust and efficient numerical software, a principal application of Fortran. Indeed, the existence of such a facility makes it possible to develop more efficient software than would otherwise be possible. Most computers nowadays have hardware based on the IEEE standard for binary floating-point arithmetic,[1] which later became an ISO standard.[2] Therefore, the Fortran exception handling features are based on the ability to test and set the five flags for floating-point exceptions that the IEEE standard specifies. However, non-IEEE computers have not been ignored; they may provide support for some of the features and the programmer is able to find out what is supported or state that certain features are essential.

Few (if any) computers support every detail of the IEEE standard. This is because considerable economies in construction and increases in execution performance are available by omitting support for features deemed to be necessary to few programmers. It was therefore decided to include inquiry facilities for the extent of support of the standard, and for the programmer to be able to state which features are essential in his or her program.

The mechanism finally chosen by the committees is based on a set of procedures for setting and testing the flags and inquiring about the features, collected in an intrinsic module called `ieee_exceptions`.

Given that procedures were being provided for the IEEE flags, it seemed sensible to provide procedures for other aspects of the IEEE standard. These are collected in a separate intrinsic module, `ieee_arithmetic`, which contains a `use` statement for `ieee_exceptions`.

To provide control over which features are essential, there is a third intrinsic module, `ieee_features` containing named constants corresponding to the features. If a named constant is accessible in a scoping unit, the corresponding feature must be available there.

18.2 The IEEE standard

In this section we explain those aspects of the IEEE standard that the reader needs to know in order to understand the features of this chapter. We do not attempt to give a complete description of the standard.

[1] IEEE 754-1985, Standard for binary floating-point arithmetic.

[2] IEC 559:1989, Binary floating-point arithmetic for microprocessor systems, now superseded by ISO/IEC/IEEE 60559:2011, Information Technology – Microprocessor Systems – Floating-Point Arithmetic.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

Two floating-point data formats are specified, one for real and one for double precision arithmetic. They are supersets of the Fortran model, repeated here (see Section 9.9.1),

$$x = 0$$

and

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}$$

where s is ± 1, p and b are integers exceeding one, e is an integer in a range $e_{\min} \le e \le e_{\max}$, and each f_k is an integer in the range $0 \le f_k < b$ except that f_1 is also nonzero. Both IEEE formats are binary, with $b = 2$. The precisions are $p = 24$ and $p = 53$, and the exponent ranges are $-125 \le e \le 128$ and $-1021 \le e \le 1024$, for real and double precision, respectively.

In addition, there are numbers with $e = e_{\min}$ and $f_1 = 0$, which are known as **denormalized**[3] numbers; note that they all have absolute values less than that returned by the intrinsic `tiny` since it considers only numbers within the Fortran model. Also, zero has a sign and both 0 and -0 have inverses, ∞ and $-\infty$. Within Fortran, -0 is treated as the same as a zero in all intrinsic operations and comparisons, but it can be detected by the `sign` function and is respected on formatted output.

The IEEE standard also specifies that some of the binary patterns that do not fit the model be used for the results of exceptional operations, such as 0/0. Such a number is known as a **NaN** (Not a Number). A NaN may be **signaling** or **quiet**. Whenever a signaling NaN appears as an operand, the invalid exception signals and the result is a quiet NaN. Quiet NaNs propagate through almost every arithmetic operation without signaling an exception.

The standard specifies four rounding modes:

nearest rounds the exact result to the nearest representable value;

to-zero rounds the exact result towards zero to the next representable value;

up rounds the exact result towards $+\infty$ to the next representable value;

down rounds the exact result towards $-\infty$ to the next representable value.

Some computers perform division by inverting the denominator and then multiplying by the numerator. The additional round-off that this involves means that such an implementation does not conform with the IEEE standard. The IEEE standard also specifies that `sqrt` returns the properly rounded value of the exact result and returns -0 for $\sqrt{-0}$. The Fortran facilities include inquiry functions for IEEE division and `sqrt`.

The presence of -0, ∞, $-\infty$, and the NaNs allows IEEE arithmetic to be closed, that is, every operation has a result. This is very helpful for optimization on modern hardware since several operations, none needing the result of any of the others, may actually be progressing in parallel. If an exception occurs, execution continues with the corresponding flag signaling, and the flag remains signaling until explicitly set quiet by the program. The flags are therefore called **sticky**.

[3] Called 'subnormal' in the 2011 standard.

There are five flags:

overflow occurs if the exact result of an operation with two normal values is too large for the data format. The stored result is ∞, `huge(x)`, `-huge(x)`, or −∞, according to the rounding mode in operation, always with the correct sign.

divide_by_zero occurs if a finite nonzero value is divided by zero. The stored result is ∞ or −∞ with the correct sign.

invalid occurs if the operation is invalid, for example, ∞×0, 0/0, or when an operand is a signaling NaN.

underflow occurs if the result of an operation with two finite nonzero values cannot be represented exactly and is too small to represent with full precision. The stored result is the best available, depending on the rounding mode in operation.

inexact occurs if the exact result of an operation cannot be represented in the data format without rounding.

The IEEE standard specifies the possibility of exceptions being trapped by user-written handlers, but this inhibits optimization and is not supported by Fortran. Instead, Fortran supports the possibility of halting program execution after an exception signals. For the sake of optimization, such halting need not occur immediately.

The IEEE standard specifies several functions that are implemented in Fortran as `ieee_copy_sign`, `ieee_logb`, `ieee_next_after`, `ieee_rem`, `ieee_rint`, `ieee_scalb`, and `ieee_unordered`, and are described in Section 18.9.3.

18.3 Access to the features

To access the features of this chapter, we recommend that the user employ `use` statements for one or more of the intrinsic modules `ieee_exceptions`, `ieee_arithmetic` (which contains a `use` statement for `ieee_exceptions`), and `ieee_features`. If the processor does not support a module accessed in a `use` statement, the compilation, of course, fails.

If a scoping unit does not access `ieee_exceptions` or `ieee_arithmetic`, the level of support is processor dependent, and need not include support for any exceptions. If a flag is signaling on entry to such a scoping unit, the processor ensures that it is signaling on exit. If a flag is quiet on entry to such a scoping unit, whether it is signaling on exit is processor dependent.

The module `ieee_features` contains the derived type

```
ieee_features_type
```

for identifying a particular feature. The only possible values objects of this type may take are those of named constants defined in the module, each corresponding to an IEEE feature. If a scoping unit has access to one of these constants, the compiler must support the feature in the scoping unit or reject the program. For example, some hardware is much faster if denormalized numbers are not supported and instead all underflowed values are flushed to zero. In such a case, the statement

```
use, intrinsic :: ieee_features, only: ieee_denormal
```

will ensure that the scoping unit is compiled with (slower) code supporting denormalized numbers. This form of the `use` statement is safer because it ensures that should there be another module with the same name, the intrinsic one is used. It is described fully in Section 9.23.

The module is unusual in that all a program ever does is to access it with `use` statements, which affect the way the code is compiled in the scoping units with access to one or more of the module's constants. There is no purpose in declaring data of type `ieee_features_type`, though it is permitted; the components of the type are private, no operation is defined for it, and only intrinsic assignment is available for it. In a scoping unit containing a `use` statement the effect is that of a compiler directive, but the other properties of `use` make the feature more powerful than would be possible with a directive.

The complete set of named constants in the module and the effect of their accessibility is:

ieee_datatype The scoping unit must provide IEEE arithmetic for at least one kind of real.

ieee_denormal The scoping unit must support denormalized numbers for at least one kind of real.

ieee_divide The scoping unit must support IEEE divide for at least one kind of real.

ieee_halting The scoping unit must support control of halting for each flag supported.

ieee_inexact_flag The scoping unit must support the inexact exception for at least one kind of real.

ieee_inf The scoping unit must support ∞ and $-\infty$ for at least one kind of real.

ieee_invalid_flag The scoping unit must support the invalid exception for at least one kind of real.

ieee_nan The scoping unit must support NaNs for at least one kind of real.

ieee_rounding The scoping unit must support control of the rounding mode for all four rounding modes on at least one kind of real.

ieee_sqrt The scoping unit must support IEEE square root for at least one kind of real.

ieee_underflow_flag The scoping unit must support the underflow exception for at least one kind of real.

Execution may be slowed on some processors by the support of some features. If `ieee_exceptions` is accessed but `ieee_features` is not accessed, the vendor is free to choose which subset to support. The processor's fullest support is provided when all of `ieee_features` is accessed:

```
use, intrinsic :: ieee_arithmetic
use, intrinsic :: ieee_features
```

but execution may then be slowed by the presence of a feature that is not needed. In all cases, inquiries about the extent of support may be made by calling the functions of Sections 18.8.2 and 18.9.2.

18.4 The Fortran flags

There are five Fortran exception flags, corresponding to the five IEEE flags. Each has a value that is either quiet or signaling. The value may be determined by the function `ieee_get_flag` (Section 18.8.3). Its initial value is quiet and it signals when the associated exception occurs in a real or complex operation. Its status may also be changed by the subroutine `ieee_set_flag` (Section 18.8.3) or the subroutine `ieee_set_status` (Section 18.8.4). Once signaling, it remains signaling unless set quiet by an invocation of the subroutine `ieee_set_flag` or the subroutine `ieee_set_status`. For invocation of an elemental procedure, it is as if the procedure were invoked once for each set of corresponding elements; if any of the invocations return with a flag signaling, it will be signaling in the caller on completion of the call.

If a flag is signaling on entry to a procedure, the processor will set it to quiet on entry and restore it to signaling on return. This allows exception handling within the procedure to be independent of the state of the flags on entry, while retaining their 'sticky' properties: within a scoping unit, a signaling flag remains signaling until explicitly set quiet. Evaluation of a specification expression may cause an exception to signal.

If a scoping unit has access to `ieee_exceptions` and references an intrinsic procedure that executes normally, the values of the overflow, divide-by-zero, and invalid flags are as on entry to the intrinsic procedure, even if one or more signals during the calculation. If a real or complex result is too large for the intrinsic procedure to handle, overflow may signal. If a real or complex result is a NaN because of an invalid operation (for example, `log(-1.0)`), invalid may signal. Similar rules apply to format processing and to intrinsic operations: no signaling flag shall be set quiet and no quiet flag shall be set signaling because of an intermediate calculation that does not affect the result.

An implementation may provide alternative versions of an intrinsic procedure; for example, one might be rather slow but be suitable for a call from a scoping unit with access to `ieee_exceptions`, while an alternative faster one might be suitable for other cases.

If it is known that an intrinsic procedure will never need to signal an exception, there is no requirement for it to be handled – after all, there is no way that the programmer will be able to tell the difference. The same principle applies to a sequence of in-line code with no invocations of `ieee_get_flag`, `ieee_set_flag`, `ieee_get_status`, `ieee_set_status`, or `ieee_set_halting`. If the code, as written, includes an operation that would signal a flag, but after execution of the sequence no value of a variable depends on that operation, whether the exception signals is processor dependent. Thus, an implementation is permitted to optimize such an operation away. For example, when `y` has the value zero, whether the code

```
x = 1.0/y
x = 3.0
```

signals divide-by-zero is processor dependent. Another example is:

```
real, parameter :: x=0.0, y=6.0
⋮
if (1.0/x == y) print *,'Hello world'
```

where the processor is permitted to discard the `if` statement since the logical expression can never be true and no value of a variable depends on it.

An exception does not signal if this could arise only during execution of code not required or permitted by the standard. For example, the statement

```
if (f(x) > 0.0) y = 1.0/z
```

must not signal divide-by-zero when both `f(x)` and `z` are zero, and the statement

```
where(a > 0.0) a = 1.0/a
```

must not signal divide-by-zero. On the other hand, when `x` has the value 1.0 and `y` has the value 0.0, the expression

```
x > 0.00001 .or. x/y > 0.00001
```

is permitted to cause the signaling of divide-by-zero.

The processor need not support the invalid, underflow, and inexact exceptions. If an exception is not supported, its flag is always quiet. The function `ieee_support_flag` (Section 18.8.2) may be used to inquire whether a particular flag is supported. If invalid is supported, it signals in the case of conversion to an integer (by assignment or an intrinsic procedure) if the result is too large to be representable.

18.5 Halting

Some processors allow control during program execution of whether to abort or continue execution after an exception has occurred. Such control is exercised by invocation of the subroutine `ieee_set_halting_mode` (Section 18.8.3). Halting is not precise and may occur any time after the exception has occurred. The function `ieee_support_halting` (Section 18.8.2) may be used to inquire whether this facility is available. The initial halting mode is processor dependent.

In a procedure other than `ieee_set_halting_mode` the processor does not change the halting mode on entry, and on return ensures that the halting mode is the same as it was on entry.

18.6 The rounding mode

Some processors support alteration of the rounding mode during execution. In this case, the subroutine `ieee_set_rounding_mode` (Section 18.9.4) may be used to alter it. The function `ieee_support_rounding` (Section 18.9.2) may be used to inquire whether this facility is available for a particular mode.

In a procedure other than `ieee_set_rounding_mode` the processor does not change the rounding mode on entry, and on return ensures that the rounding mode is the same as it was on entry. This means that the rounding mode is unaffected by any procedure reference except for a call to `ieee_set_rounding_mode` itself, including the case of a function invoked within an expression.

Note that the value of a literal constant is not affected by the rounding mode.

18.7 The underflow mode

Some processors support alteration of the underflow mode during execution, that is, whether small values are represented as denormalized values or are set to zero. The reason is likely to be that such a processor executes much faster without denormalized values. The underflow mode is said to be **gradual** if denormalized values are employed. If the underflow mode may be altered at run time, the subroutine `ieee_set_underflow_mode` (Section 18.9.4) may be used to alter it. The function `ieee_support_underflow_control` (Section 18.9.2) may be used to inquire whether this facility is available for a particular kind of real.

In a procedure other than `ieee_set_underflow_mode` the processor does not change the underflow mode on entry, and on return ensures that it is the same as it was on entry.

18.8 The module ieee_exceptions

When the module `ieee_exceptions` is accessible, the overflow and divide-by-zero flags are supported in the scoping unit for all available kinds of real and complex data. This minimal level of support has been designed to be possible also on a non-IEEE computer. Which other exceptions are supported may be determined by the function `ieee_support_flag`, see Section 18.8.2. Whether control of halting is supported may be determined by the function `ieee_support_halting`, see Section 18.8.2. The extent of support of the other exceptions may be influenced by the accessibility of the named constants `ieee_inexact_flag`, `ieee_invalid_flag`, and `ieee_underflow_flag` of the module `ieee_features`, see Section 18.3.

The module contains two derived types (Section 18.8.1), named constants of these types (Section 18.8.1), and a collection of generic procedures (Sections 18.8.2, 18.8.3, and 18.8.4). None of the procedures is permitted as an actual argument.

18.8.1 Derived types

The module `ieee_exceptions` contains two derived types:

`ieee_flag_type` for identifying a particular exception flag. The only values that can be taken by objects of this type are those of named constants defined in the module,

```
ieee_overflow   ieee_divide_by_zero  ieee_invalid
ieee_underflow  ieee_inexact
```

and these are used in the module to define the named array constants:

```
type(ieee_flag_type), parameter ::                          &
ieee_usual(3) =                                             &
    (/ieee_overflow, ieee_divide_by_zero, ieee_invalid/), &
ieee_all(5) = (/ieee_usual, ieee_underflow, ieee_inexact/)
```

These array constants are convenient for inquiring about the state of several flags at once by using elemental procedures. Besides convenience, such elemental calls may be more efficient than a sequence of calls for single flags.

ieee_status_type for saving the current floating-point status. It includes the values of all the flags supported, and also the current rounding mode, if dynamic control of rounding is supported, and the halting mode, if dynamic control of halting is supported.

The components of both types are private. No operation is defined for them and only intrinsic assignment is available for them.

18.8.2 Inquiring about IEEE exceptions

The module `ieee_exceptions` contains two functions, both of which are pure. Their argument `flag` must be of type `type(ieee_flag_type)` with one of the values `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_inexact`, and `ieee_underflow`. The inquiries are about the support for kinds of reals and the same level of support is provided for the corresponding kinds of complex type.

ieee_support_flag(flag) or **ieee_support_flag(flag,x)** returns `.true.` if the processor supports the exception `flag` for all reals or for reals of the same kind type parameter as the real argument `x`, respectively; otherwise, it returns `.false.`.

ieee_support_halting (flag) returns `.true.` if the processor supports the ability to change the mode by `call ieee_set_halting_mode(flag, halting)`; otherwise, it returns `.false.`.

18.8.3 Subroutines for the flags and halting modes

The module `ieee_exceptions` contains the following elemental subroutines:

call ieee_get_flag (flag, flag_value) where:

flag is of type `type(ieee_flag_type)`. It specifies a flag.

flag_value is of type default logical and has intent `out`. If the value of `flag` is `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_underflow`, or `ieee_inexact`, `flag_value` is given the value true if the corresponding exception flag is signaling and false otherwise.

call ieee_get_halting_mode (flag, halting) where:

flag is of type `type(ieee_flag_type)`. It must have one of the values `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_underflow`, or `ieee_inexact`.

halting is of type default logical and has intent `out`. If the exception specified by `flag` will cause halting, `halting` is given the value true; otherwise, it is given the value false.

Elemental subroutines would not be appropriate for the corresponding 'set' actions since an invocation might ask for a flag or mode to be set more than once. The module therefore contains the following subroutines that are pure but not elemental:

`call ieee_set_flag (flag, flag_value)` where:

`flag` is of type `type(ieee_flag_type)`. It may be scalar or array valued. If it is an array, no two elements may have the same value.

`flag_value` is of type default logical. It must be conformable with `flag`. Each flag specified by `flag` is set to be signaling if the corresponding `flag_value` is true, and to be quiet if it is false.

`call ieee_set_halting_mode (flag, halting)` which may be called only if the value returned by `ieee_support_halting(flag)` is true:

`flag` is of type `type(ieee_flag_type)`. It may be scalar or array valued. If it is an array, no two elements may have the same value.

`halting` is of type default logical. It must be conformable with `flag`. Each exception specified by `flag` will cause halting if the corresponding value of `halting` is true and will not cause halting if the value is false.

18.8.4 Subroutines for the whole of the floating-point status

The module `ieee_exceptions` contains the following non-elemental subroutines:

`call ieee_get_status (status_value)` where:

`status_value` is scalar and of type `type(ieee_status_type)`, and has intent `out`. It returns the floating-point status, including all the exception flags, the rounding mode, and the halting mode.

`call ieee_set_status (status_value)` where:

`status_value` is scalar and of type `type(ieee_status_type)`. Its value must have been set in a previous invocation of `ieee_get_status`. The floating-point status, including all the exception flags, the rounding mode, and the halting mode, is reset to what it was then.

Figure 18.1 Performing a subsidiary calculation with an independent set of flags.

```
use, intrinsic         :: ieee_exceptions
type(ieee_status_type) :: status_value
⋮
call ieee_get_status(status_value)   ! Get the flags
call ieee_set_flag(ieee_all,.false.) ! Set the flags quiet
⋮  ! Calculation involving exception handling
call ieee_set_status(status_value)   ! Restore the flags
```

These subroutines have been included for convenience and efficiency when a subsidiary calculation is to be performed and one wishes to resume the main calculation with exactly

the same environment, as shown in Figure 18.1. There are no facilities for finding directly the value held within such a variable of a particular flag, rounding mode, or halting mode.

18.9 The module ieee_arithmetic

The module `ieee_arithmetic` behaves as if it contained a `use` statement for the module `ieee_exceptions`, so all the features of `ieee_exceptions` are also features of `ieee_arithmetic`.

The module contains two derived types (Section 18.9.1), named constants of these types (Section 18.9.1), and a collection of generic procedures (Sections 18.9.2, 18.9.3, 18.9.4, and 18.9.5). None of the procedures is permitted as an actual argument.

18.9.1 Derived types

The module `ieee_arithmetic` contains two derived types:

`ieee_class_type` for identifying a class of floating-point values. The only values objects of this type may take are those of the named constants defined in the module,

```
ieee_signaling_nan     ieee_quiet_nan
ieee_negative_inf      ieee_negative_normal
ieee_negative_denormal ieee_negative_zero
ieee_positive_zero     ieee_positive_denormal
ieee_positive_normal   ieee_positive_inf
```

with obvious meanings, and `ieee_other_value` for any cases that cannot be so identified, for example if an unformatted file were written with gradual underflow enabled and read with it disabled.

`ieee_round_type` for identifying a particular rounding mode. The only possible values objects of this type may take are those of the named constants defined in the module,

```
ieee_nearest ieee_to_zero
ieee_up      ieee_down
```

for the IEEE modes and `ieee_other` for any other mode.

The components of both types are private. The only operations defined for them are == and /= for comparing values of one of the types; they return a value of type default logical. Intrinsic assignment is also available.

18.9.2 Inquiring about IEEE arithmetic

The module `ieee_arithmetic` contains the following functions, all of which are pure. The inquiries are about the support of reals, and the same level of support is provided for the corresponding kinds of complex type. The argument `x` may be a scalar or an array. All but `ieee_support_rounding` are inquiry functions.

ieee_support_datatype( ) or **ieee_support_datatype(x)** returns .true. if the processor supports IEEE arithmetic for all reals or for reals of the same kind type parameter as the real argument x, respectively. Otherwise, it returns .false.. Complete conformance with the IEEE standard is not required for .true. to be returned, but the normalized numbers must be exactly those of IEEE single or IEEE double; the binary arithmetic operators +, –, and * must be implemented with at least one of the IEEE rounding modes; and the functions abs, ieee_copy_sign, ieee_scalb, ieee_logb, ieee_next_after, ieee_rem, and ieee_unordered must implement the corresponding IEEE functions.

ieee_support_denormal( ) or **ieee_support_denormal(x)** returns .true. if the processor supports the IEEE denormalized numbers for all reals or for reals of the same kind type parameter as the real argument x, respectively. Otherwise, it returns .false..

ieee_support_divide( ) or **ieee_support_divide(x)** returns .true. if the processor supports divide with the accuracy specified by the IEEE standard for all reals or for reals of the same kind type parameter as the real argument x, respectively. Otherwise, it returns .false..

ieee_support_inf( ) or **ieee_support_inf(x)** returns .true. if the processor supports the IEEE infinity facility for all reals or for reals of the same kind type parameter as the real argument x, respectively. Otherwise, it returns .false..

ieee_support_io( ) or **ieee_support_io(x)** returns .true. if the results of formatted input/output satisfy the requirements of the IEEE standard for all four IEEE rounding modes for all reals or for reals of the same kind type parameter as the real argument x, respectively. Otherwise, it returns .false..

ieee_support_nan( ) or **ieee_support_nan(x)** returns .true. if the processor supports the IEEE NaN facility for all reals or for reals of the same kind type parameter as the real argument x, respectively. Otherwise, it returns .false..

ieee_support_rounding (round_value) or **ieee_support_rounding (round_value,x)** for a round_value of the type ieee_round_type returns .true. if the processor supports that rounding mode for all reals or for reals of the same kind type parameter as the argument x, respectively. Otherwise, it returns .false.. Here, support includes the ability to change the mode by the invocation

```
call ieee_set_rounding_mode (round_value)
```

ieee_support_sqrt( ) or **ieee_support_sqrt(x)** returns .true. if sqrt implements IEEE square root for all reals or for reals of the same kind type parameter as the real argument x, respectively. Otherwise, it returns .false..

ieee_support_standard( ) or **ieee_support_standard(x)** returns .true. if the processor supports all the IEEE facilities defined in this chapter for all reals or for reals of the same kind type parameter as the real argument x, respectively. Otherwise, it returns .false..

ieee_support_underflow_control( ) or
ieee_support_underflow_control(x) returns .true. if the processor supports control of the underflow mode for all reals or for reals of the same kind type parameter as the real argument x, respectively. Otherwise, it returns .false..

18.9.3 Elemental functions

The module ieee_arithmetic contains the following elemental functions for the reals x and y for which the values of ieee_support_datatype(x) and ieee_support_datatype(y) are true. If x or y is an infinity or a NaN, the behaviour is consistent with the general rules of the IEEE standard for arithmetic operations. For example, the result for an infinity is constructed as the limiting case of the result with a value of arbitrarily large magnitude, when such a limit exists.

ieee_class (x) is of type type(ieee_class_type) and returns the IEEE class of the real argument x. The possible values are explained in Section 18.9.1.

ieee_copy_sign (x, y) returns a real with the same type parameter as x, holding the value of x with the sign of y. This is true even for the IEEE special values, such as NaN and ∞ (on processors supporting such values).

ieee_is_finite (x) returns the value .true. if ieee_class (x) has one of the values

```
ieee_negative_normal   ieee_negative_denormal
ieee_negative_zero     ieee_positive_zero
ieee_positive_denormal ieee_positive_normal
```

and .false. otherwise.

ieee_is_nan (x) returns the value .true. if the value of x is an IEEE NaN and .false. otherwise.

ieee_is_negative (x) returns the value .true. if ieee_class (x) has one of the values

```
ieee_negative_normal ieee_negative_denormal
ieee_negative_zero   ieee_negative_inf
```

and .false. otherwise.

ieee_is_normal (x) returns the value .true. if ieee_class (x) has one of the values

```
ieee_negative_normal  ieee_negative_zero
ieee_positive_zero    ieee_positive_normal
```

and .false. otherwise.

ieee_logb (x) returns a real with the same type parameter as x. If x is neither zero, infinity, nor NaN, the value of the result is the unbiased exponent of x, that is, exponent(x)-1. If x==0, the result is $-\infty$ if ieee_support_inf(x) is true and -huge(x); otherwise, ieee_divide_by_zero signals. If ieee_support_inf(x) is true and x is infinite, the result is +infinity. If ieee_support_nan(x) is true and x is a NaN, the result is a NaN.

ieee_next_after (x, y) returns a real with the same type parameter as x. If x==y, the result is x, without an exception ever signaling. Otherwise, the result is the neighbour of x in the direction of y. The neighbours of zero (of either sign) are both nonzero. Overflow is signaled when x is finite but ieee_next_after (x, y) is infinite; underflow is signaled when ieee_next_after (x, y) is denormalized; in both cases, ieee_inexact signals.

ieee_rem (x, y) returns a real with the type parameter of whichever argument has the greater precision and value exactly x-y*n, where n is the integer nearest to the exact value x/y; whenever $|n - \mathrm{x/y}| = 1/2$, n is even. If the result value is zero, the sign is that of x.

ieee_rint (x) returns a real with the same type parameter as x whose value is that of x rounded to an integer value according to the current rounding mode.

ieee_scalb (x, i) returns a real with the same type parameter as x whose value is 2^{i}x if this is within the range of normal numbers. If 2^{i}x is too large, ieee_overflow signals; if ieee_support_inf(x) is true, the result value is infinity with the sign of x; otherwise, it is sign(huge(x),x). If 2^{i}x is too small and cannot be represented exactly, ieee_underflow signals; the result is the nearest representable number with the sign of x.

ieee_unordered (x, y) returns .true. if x or y is a NaN or both are, and .false. otherwise.

ieee_value (x, class) returns a real with the same type parameter as x and a value specified by class. The argument class is of type type(ieee_class_type) and may have value

- ieee_signaling_nan or ieee_quiet_nan if ieee_support_nan(x) is true;
- ieee_negative_inf or ieee_positive_inf if ieee_support_inf(x) is true;
- ieee_negative_denormal or ieee_positive_denormal if the value of ieee_support_denormal(x) is true; or
- ieee_negative_normal, ieee_negative_zero, ieee_positive_zero, or ieee_positive_normal.

Although in most cases the value is processor dependent, it does not vary between invocations for any particular kind type parameter of x and value of class.

18.9.4 Non-elemental subroutines

The module `ieee_arithmetic` contains the following non-elemental subroutines:

`call ieee_get_rounding_mode (round_value)` where:

> **`round_value`** is scalar, of type `type(ieee_round_type)`, and has intent `out`. It returns the floating-point rounding mode, with value `ieee_nearest`, `ieee_to_zero`, `ieee_up`, or `ieee_down` if one of the IEEE modes is in operation, and `ieee_other` otherwise.

`call ieee_get_underflow_mode (gradual)` where:

> **`gradual`** is scalar, of type default `logical`, and has intent `out`. It returns `.true.` if gradual underflow is in effect, and `.false.` otherwise.

`call ieee_set_rounding_mode (round_value)` where:

> **`round_value`** is scalar, of type `type(ieee_round_type)`. It specifies the mode to be set.
>
> The subroutine must not be called unless the value of `ieee_support_rounding (round_value, x)` is true for some `x` such that the value of `ieee_support_datatype(x)` is true.

`call ieee_set_underflow_mode (gradual)` where:

> **`gradual`** is scalar, of type default `logical`. If its value is `.true.`, gradual underflow comes into effect; otherwise gradual underflow ceases to be in effect.
>
> The subroutine must not be called unless `ieee_support_underflow_control (x)` is true for some `x`.

Figure 18.2 Store the rounding mode, perform a calculation with another mode, and restore the previous mode.

```
use, intrinsic :: ieee_arithmetic
type(ieee_round_type) round_value
⋮
call ieee_get_rounding_mode(round_value) ! Store the rounding mode
call ieee_set_rounding_mode(ieee_nearest)
⋮ ! Calculation with round to nearest
call ieee_set_rounding_mode(round_value) ! Restore the rounding mode
```

The example in Figure 18.2 shows the use of these subroutines to store the rounding mode, perform a calculation with round to nearest, and restore the rounding mode.

18.9.5 Transformational function for kind value

The module `ieee_arithmetic` contains the following transformational function that is permitted in a constant expression (Section 8.4):

`ieee_selected_real_kind (`*`[p] [, r] [,`* `radix]``)` is similar to the function `selected_real_kind` (Section 9.9.4) except that the result is the kind value of a real `x` for which `ieee_support_datatype(x)` is true.

18.10 Examples

18.10.1 Dot product

Figure 18.3 Module for the dot product of two real rank-one arrays.

```
module dot
 ! The caller must ensure that exceptions do not cause halting.
   use, intrinsic :: ieee_exceptions
   implicit none
   private        :: mult
   logical        :: dot_error = .false.
   interface operator(.dot.)
      module procedure mult
   end interface
contains
   real function mult(a, b)
      real, intent(in) :: a(:), b(:)
      integer          :: i
      logical          :: overflow
      if (size(a)/=size(b)) then
         dot_error = .true.
         return
      end if
! The processor ensures that ieee_overflow is quiet
      mult = 0.0
      do i = 1, size(a)
         mult = mult + a(i)*b(i)
      end do
      call ieee_get_flag(ieee_overflow, overflow)
      if (overflow) dot_error = .true.
   end function mult
end module dot
```

Our first example, Figure 18.3, is of a module for the dot product of two real arrays of rank one. It contains a logical scalar `dot_error`, which acts as an error flag. If the sizes of

the arrays are different, an immediate return occurs with `dot_error` true. If overflow occurs during the actual calculation, the overflow flag will signal and `dot_error` is set true. If all is well, its value is unchanged.

18.10.2 Calling alternative procedures

Suppose the function `fast_inv` is code for matrix inversion that 'lives dangerously' and may cause a condition to signal. The alternative function `slow_inv` is far less likely to cause a condition to signal, but is much slower. The following code, Figure 18.4, tries `fast_inv` and, if necessary, makes another try with `slow_inv`. If this still fails, a message is printed and the program stops. Note, also, that it is important to set the flags quiet before the second try. The state of all the flags is stored and restored.

Figure 18.4 Try a fast algorithm and, if necessary, try again with a slower but more reliable algorithm.

```
use, intrinsic :: ieee_exceptions
use, intrinsic :: ieee_features, only: ieee_invalid_flag
! The other exceptions of ieee_usual (ieee_overflow and
! ieee_divide_by_zero) are always available with ieee_exceptions
type(ieee_status_type) :: status_value
logical, dimension(3)  :: flag_value
⋮

call ieee_get_status(status_value)
call ieee_set_halting_mode(ieee_usual,.false.) ! Needed in case the
!                default on the processor is to halt on exceptions.
call ieee_set_flag(ieee_usual,.false.)         ! Elemental
! First try the "fast" algorithm for inverting a matrix:
matrix1 = fast_inv(matrix) ! This must not alter matrix.
call ieee_get_flag(ieee_usual, flag_value)     ! Elemental
if (any(flag_value)) then
! "Fast" algorithm failed; try "slow" one:
   call ieee_set_flag(ieee_usual,.false.)
   matrix1 = slow_inv(matrix)
   call ieee_get_flag(ieee_usual, flag_value)
   if (any(flag_value)) then
      write (*, *) 'Cannot invert matrix'
      stop
   end if
end if
call ieee_set_status(status_value)
```

18.10.3 Calling alternative in-line code

This example, Figure 18.5, is similar to the inner part of the previous one, but here the code for matrix inversion is in line, we know that only overflow can signal, and the transfer is made more precise by adding extra tests of the flag.

Figure 18.5 As for Figure 18.4 but with in-line code.

```
use, intrinsic :: ieee_exceptions
logical        :: flag_value
⋮

call ieee_set_halting_mode(ieee_overflow,.false.)
call ieee_set_flag(ieee_overflow,.false.)
! First try a fast algorithm for inverting a matrix.
do k = 1, n
   ⋮
   call ieee_get_flag(ieee_overflow, flag_value)
   if (flag_value) exit
end do
if (flag_value) then
! Alternative code which knows that k-1 steps have
! executed normally.
⋮

end if
```

18.10.4 Reliable hypotenuse function

The most important use of a floating-point exception handling facility is to make possible the development of much more efficient software than is otherwise possible. The code in Figure 18.6 for the 'hypotenuse' function, $\sqrt{x^2+y^2}$, illustrates the use of the facility in developing efficient software.

An attempt is made to evaluate this function directly in the fastest possible way. This will work almost every time, but if an exception occurs during this fast computation, a safe but slower way evaluates the function. This slower evaluation may involve scaling and unscaling, and in (very rare) extreme cases this unscaling can cause overflow (after all, the true result might overflow if `x` and `y` are both near the overflow limit). If the overflow or underflow flag is signaling on entry, it is reset on return by the processor so that earlier exceptions are not lost.

Figure 18.6 A reliable hypotenuse function.

```
real function hypot(x, y)

! In rare circumstances this may lead to the signaling of
! ieee_overflow.
!
   use, intrinsic :: ieee_exceptions
   use, intrinsic :: ieee_features, only: ieee_underflow_flag
! ieee_overflow is always available with ieee_exceptions

   implicit none
   real                  :: x, y
   real                  :: scaled_x, scaled_y, scaled_result
   logical, dimension(2) :: flags
   type(ieee_flag_type), parameter, dimension(2) ::        &
          out_of_range = (/ ieee_overflow, ieee_underflow /)
   intrinsic :: sqrt, abs, exponent, max, digits, scale
! The processor clears the flags on entry
   call ieee_set_halting_mode(out_of_range, .false.) ! Needed in
!     case the default on the processor is to halt on exceptions.
! Try a fast algorithm first
   hypot = sqrt( x**2 + y**2 )
   call ieee_get_flag(out_of_range, flags)
   if ( any(flags) ) then
     call ieee_set_flag(out_of_range, .false.)
     if ( x==0.0 .or. y==0.0 ) then
       hypot = abs(x) + abs(y)
     else if ( 2*abs(exponent(x)-exponent(y)) > digits(x)+1 ) then
       hypot = max( abs(x), abs(y) )! We can ignore one of x and y
     else     ! Scale so that abs(x) is near 1
       scaled_x = scale( x, -exponent(x) )
       scaled_y = scale( y, -exponent(x) )
       scaled_result = sqrt( scaled_x**2 + scaled_y**2 )
       hypot = scale(scaled_result, exponent(x)) ! May cause
     end if                                      ! overflow
   end if
! The processor resets any flag that was signaling on entry
end function hypot
```

18.10.5 Access to IEEE arithmetic values

The program in Figure 18.7 illustrates how the `ieee_arithmetic` module can be used to test for special IEEE values. It repeatedly doubles `a` and halves `b`, testing for overflowed, denormalized, and zero values. It uses `ieee_set_halting_mode` to prevent halting. The beginning and end of a sample output are shown. Note the warning messages; the processor is required to produce some such output if any exceptions are signaling at termination.

Figure 18.7 Test for overflowed, denormalized, and zero values.

```
program test
   use ieee_arithmetic; use ieee_features
   real    :: a=1.0, b=1.0
   integer :: i
   call ieee_set_halting_mode(ieee_overflow, .false.)
   do i = 1,1000
      a = a*2.0
      b = b/2.0
      if (.not. ieee_is_finite(a)) then
         write (*, *) '2.0**', i, ' is infinite'
         a = 0.0
      end if
      if (.not. ieee_is_normal(b)) &
         write (*, *) '0.5**', i, ' is denormal'
      if (b==0.0) exit
   end do
   write (*, *) '0.5**', i, ' is zero'
end program test
```

```
 0.5** 127  is denormal
 2.0** 128  is infinite
 0.5** 128  is denormal
 0.5** 129  is denormal
 :
 0.5** 148  is denormal
 0.5** 149  is denormal
 0.5** 150  is zero
Warning: Floating overflow occurred during execution
Warning: Floating underflow occurred during execution
```

19. Interoperability with C

19.1 Introduction

Fortran provides a standardized mechanism for interoperating with C. Clearly, any entity involved must be such that equivalent declarations of it may be made in the two languages. This is enforced within the Fortran program by requiring all such entities to be *interoperable*. We will explain in turn what this requires for types, variables, and procedures. They are all requirements on the syntax so that the compiler knows at compile time whether an entity is interoperable. We continue with examining interoperability for global data and then discuss some examples. We conclude with a syntax for defining sets of integer constants that is useful in this context.

19.2 Interoperability of intrinsic types

There is an intrinsic module named `iso_c_binding` that contains named constants of type default integer holding kind type parameter values for intrinsic types. Their names are shown in Table 19.1, together with the corresponding C types. The processor is required to support only `int`. Lack of support is indicated with a negative value of the constant. If the value is positive, it indicates that the Fortran type and kind type parameter interoperate with the corresponding C type.

The negative values are as follows. For the integer types, the value is -1 if there is such a C type but no interoperating Fortran kind or -2 if there is no such C type. For the real types, the value is -1 if the C type does not have a precision equal to the precision of any of the Fortran real kinds, -2 if the C type does not have a range equal to the range of any of the Fortran real kinds, -3 if the C type has neither the precision nor range of any of the Fortran real kinds, and equal to -4 if there is no interoperating Fortran kind for other reasons. The values of `c_float_complex`, `c_double_complex`, and `c_long_double_complex` are the same as those of `c_float`, `c_double`, and `c_long_double`, respectively. For logical, the value of `c_bool` is -1 if there is no Fortran kind corresponding to the C type `_Bool`. For character, the value of `c_char` is -1 if there is no Fortran kind corresponding to the C type `char`.

For character type, interoperability also requires that the length type parameter be omitted or be specified by a constant expression whose value is 1. The following named constants (with the obvious meanings) are provided: `c_null_char`, `c_alert`, `c_backspace`, `c_form_feed`, `c_new_line`, `c_carriage_return`, `c_horizontal_tab`, `c_vertical_tab`.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

They are all of type character with length one and kind `c_char` (or default kind if `c_char` has the value -1).

Table 19.1. Named constants for interoperable kinds of intrinsic Fortran types.

Type	Named constant	C type or types
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_least16_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	c_float_complex	float _Complex
	c_double_complex	double _Complex
	c_long_double_complex	long double _Complex
logical	c_bool	_Bool
character	c_char	char

19.3 Interoperability with C pointer types

For interoperating with C pointers (which are just addresses), the module contains the derived types `c_ptr` and `c_funptr` that are interoperable with C object and function pointer types,

respectively. Their components are private. There are named constants `c_null_ptr` and `c_null_funptr` for the corresponding null values of C.

The module also contains the following procedures, none of which are pure:

`c_loc (x)` is a function that returns a scalar of type `c_ptr` that holds the C address of its argument. The argument must be a pointer or have the `target` attribute. It must either be a variable with interoperable type and kind type parameters, or be a scalar non-polymorphic variable with no length type parameters. If it is an array, it must be contiguous and have nonzero size. It shall not be a zero-length string. If it is allocatable, it must be allocated. If it is a pointer it must be associated. It must not be coindexed.

`c_funloc (x)` is a function that returns the C address of a procedure. The argument `x` is permitted to be a procedure that is interoperable (see Section 19.9) or a pointer associated with such a procedure.

`c_associated (c_ptr1`*`[`*`, c_ptr2`*`]`*`)` is a function for scalars of type `c_ptr` or for scalars of type `c_funptr`. It returns a default logical scalar. It has the value false if `c_ptr1` is a C null pointer or if `c_ptr2` is present with a different value; otherwise, it has the value true.

`c_f_pointer (cptr, fptr`*`[`***`, shape`***`]`***`)`** is a subroutine where

`cptr` is a scalar of type `c_ptr` with intent `in`. Its value is either

i) the C address of an interoperable data entity; or

ii) the result of a reference to `c_loc` with a non-interoperable argument.

It must not be the C address of a Fortran variable that does not have the `target` attribute.

`fptr` is a pointer with intent `out`.

i) If `cptr` is the C address of an interoperable entity, `fptr` must be a data pointer of the type and type parameters of the entity and it becomes pointer associated with the target of `cptr`. If it is an array, its shape is specified by `shape` and each lower bound is 1.

ii) If `cptr` was returned by a call of `c_loc` with a non-interoperable argument `x`, `fptr` must be a non-polymorphic scalar pointer of the type and type parameters of `x`. `x` or its target if it is a pointer shall not have since been deallocated or have become undefined due to execution of a `return` or an `end` statement. `fptr` becomes pointer-associated with `x` or its target.

`shape` (optional) is a rank-one array of type integer with intent `in`. If present, its size is equal to the rank of `fptr`. It must be present if `fptr` is an array.

`c_f_procpointer (cptr, fptr)` is a subroutine where

`cptr` is a scalar of type `c_funptr` with intent `in`. Its value is the C address of a procedure that is interoperable.

`fptr` is a procedure pointer with intent `out`. Its interface must be interoperable with the target of `cptr` and it becomes pointer-associated with that target.

A Fortran pointer or allocatable variable, and most Fortran arrays, do not interoperate directly with any C entity because C does not have quite the same concepts; for example, unlike a Fortran array pointer, a C array pointer cannot describe a discontiguous array section.[1] However, this does not prevent such entities being passed to C via argument association since Fortran compilers already perform copy-in copy-out when this is necessary. Also, the function `c_loc` may be used to obtain the C address of an allocated allocatable array, which is useful if the C part of the program wishes to maintain a pointer to this array. Similarly, the address of an array allocated in C may be passed to Fortran and `c_f_pointer` used to construct a Fortran pointer whose target is the C array. There is an illustration of this in Section 19.11.

Case ii) of `c_loc` allows the C program to receive a pointer to a Fortran scalar that is not interoperable. It is not intended that any use of it be made within C except to pass it back to Fortran, where `c_f_pointer` is available to reconstruct the Fortran pointer. There is an illustration of this in Section 19.12.

19.4 Interoperability of derived types

For a derived type to be interoperable, it must have the `bind` attribute:

```
type, bind(c) :: mytype
⋮
end type mytype
```

It must not be a sequence type (Appendix A.2), have type parameters, have the `extends` attribute (Section 15.2), or have any type-bound procedures (Section 15.8). Each component must have interoperable type and type parameters, must not be a zero-sized array, must not be a pointer, and must not be allocatable.

These restrictions allow the type to interoperate with a C structure type that has the same number of components. The components correspond by position in their definitions. Each Fortran component must be interoperable with the corresponding C component. Here is a simple example:

```
typedef struct
   int m, n;
   float r;
 myctype;
```

is interoperable with

```
use, intrinsic :: iso_c_binding
type, bind(c) :: myftype
   integer(c_int) :: i, j
   real(c_float) :: s
end type myftype
```

[1] This deficiency is remedied in Fortran 2018 with the C descriptor, see Chapter 21.

The name of the type and the names of the components are not significant for interoperability. If two equivalent definitions of an interoperable derived type are made in separate scoping units, they interoperate with the same C type (but it is usually preferable to define one type in a module and access it by `use` statements).

No Fortran type is interoperable with a C union type, a C structure type that contains a bit field, or a C structure type that contains a flexible array member.

19.5 Shape and character length disagreement

Before discussing interoperability of variables, we need to introduce the concept of associating an array actual argument with a dummy array whose rank and bounds are declared afresh. An important case concerns the **assumed-size** array, which is a dummy array whose size is assumed from the size of the corresponding actual argument.

The concept dates from Fortran 77, and is usually implemented by passing just the address of the first element of the array. It provides a good match for C semantics, but we do not recommend its use in pure Fortran programs because it is error-prone compared with the assumed-shape array, allocatable arrays, and array sections, where the whole shape is passed and compilers can check for valid indices when in debugging mode. Because the standard allows only the address of the actual argument to be passed, and requires that only the address be passed for an interoperable procedure, it might be better to call the dummy argument an unknown-size array.

For an assumed-size array, the final upper bound is declared with an asterisk. All the bounds must be given in a declaration that is as for an explicit-shape array except for the asterisk final upper bound. The bounds of the corresponding actual argument are ignored. The elements of the assumed-size array, in array-element order, are associated with the elements of the corresponding actual argument, in array-element order. We illustrate this in Figure 19.1, where the array elements `aa(1,1)`, `aa(2,1)`, `aa(1,2)`, ... are associated with `da(1)`, `da(2)`, `da(3)`, ..., and the array elements `ab(1)`, `ab(2)`, `ab(3)`, ... are associated with `db(1,1)`, `db(2,1)`, `db(1,2)`, ...

Figure 19.1 Passing arrays to assumed-size arrays.

```
real :: aa(2,3), ab(6)
⋮
call sub (a, b)
⋮
subroutine sub(a, b)
real :: da(*), db(2,*)
⋮
```

Because an assumed-size array has no upper bound in its last dimension, it does not have a shape and therefore must not be used as a whole array, except as an argument to a procedure that does not require its shape. However, if an array section of it is formed with an explicit upper bound in the last dimension, this has a shape and may be used as a whole array.

Because it would require an action for every element of an array of unknown size, an assumed-size array is not permitted to have intent `out` if it is polymorphic, of a derived type with default initialization or an ultimate allocatable component, or of a finalizable type.

It is also permitted to associate an element of an array that is neither a pointer, of assumed size, nor polymorphic (such an array is always contiguous) with an assumed-size array. This is interpreted as associating the subarray consisting of all the elements of the array from the given element onwards in array-element order with the assumed-size array. Figure 19.2 illustrates this. Here, only the last 49 elements of `a` are available to `sub`, as the first array

Figure 19.2 Passing an array element to an assumed-size array.

```
real :: a(100)
⋮
call sub (a(52))
⋮
subroutine sub(b)
real :: b(*)
⋮
```

element of `a` which is passed to `sub` is `a(52)`. Within `sub`, this element is referenced as `b(1)` and it is illegal to address `b(50)` in any way, as that would be beyond the declared length of `a` in the calling procedure. In Fortran 77 this mechanism provided a limited form of array section. We do not recommend its use in modern Fortran, which has the far better feature of the array section.

These mechanisms may also be used when the dummy argument is an explicit-shape array. The size of the dummy array is not permitted to be greater than the size of the actual array or the part of it from the given element onwards, but this is rarely checked by compilers.

In the case of default character type, agreement of character length is not required. For a scalar dummy argument of character length *len*, the actual argument may have a greater character length and its leftmost *len* characters are associated with the dummy argument. For example, if `chasub` has a single dummy argument of character length 1,

```
call chasub(word(3:4))
```

is a valid `call` statement. For a dummy argument that is an array, the restriction is on the total number of characters in the array. An array element or array element substring is regarded as a sequence of characters from its first character to the last character of the array. The size is the number of characters in the sequence divided by the character length of the dummy argument.

Shape or character length disagreement cannot occur when a dummy argument is assumed shape (by definition, the shape is assumed from the actual argument). It can occur for explicit-shape and assumed-size arrays. Implementations usually receive explicit-shape and assumed-size arrays in contiguous storage, but permit any uniform spacing of the elements of an assumed-shape array. They will need to make a copy of any array argument that is not stored contiguously (for example, the section `a(1:10:2)`), unless the dummy argument is assumed shape.

The rules on character length disagreement include `character(kind=c_char)` (which will often be the same as default character) and treat any other scalar actual argument of type default character or `character(kind=c_char)` as if it were an array of size one. This includes the case where the argument is an element of an assumed-shape array or an array pointer, or a subobject thereof; note that just that element or subobject is passed, not the rest of the array.

When a procedure is invoked through a generic name, as a defined operation, or as a defined assignment, rank agreement between the actual and the dummy arguments is required. In this case, only a scalar dummy argument may be associated with a scalar actual argument.

19.6 Interoperability of variables

A scalar Fortran variable is interoperable if it is of interoperable type and type parameters, and is neither a pointer nor allocatable. It is interoperable with a C scalar if the Fortran type and type parameters are interoperable with the C type.

An array Fortran variable is interoperable if its size is nonzero, it is of interoperable type and type parameters, and it is of explicit shape or assumed size.

For a Fortran array of rank one to interoperate with a C array, the Fortran array elements must be interoperable with the C array elements. If the Fortran array is of explicit size, the C array must have the same size. If the Fortran array is of assumed size, the C array must not have a specified size.

A Fortran array `a` of rank greater than one and of shape $(e_1, e_2, \ldots, e_r)$ is interoperable with a C array of size e_r with elements that are interoperable with a Fortran array of the same type as `a` and of shape $(e_1, e_2, \ldots, e_{r-1})$. For ranks greater than two, this rule is applied recursively. For example, the Fortran arrays declared as

```
integer(c_int) :: fa(18, 3:7), fb(18, 3:7, 4)
```

are interoperable with C arrays declared as

```
int ca[5][18], cb[4][5][18];
```

and the elements correspond. Note that the subscript order is reversed.

An assumed-size Fortran array of rank greater than one is interoperable with a C array of unspecified size if its elements are related to the Fortran array in the same way as in the explicit-size case. For example, the Fortran arrays declared as

```
integer(c_int) :: fa(18, *), fb(18, 3:7, *)
```

are interoperable with C arrays declared as

```
int ca[ ][18], cb[ ][5][18];
```

19.7 Function c_sizeof

The intrinsic module `iso_c_binding` also contains the inquiry function `c_sizeof`. It provides similar functionality to that of the `sizeof` operator in C.

`c_sizeof (x)` If `x` is scalar, this returns the value that the companion processor returns for the C `sizeof` operator applied to an object of a type that interoperates with the type and type parameters of `x`. If `x` is an array, the result is the value returned for an element of the array multiplied by its number of elements. `x` must be interoperable (see Section 19.6), and is not permitted to be an assumed-size array (see Section 19.5).

For example,

```
use iso_c_binding
integer(c_int64_t) x
print *, c_sizeof(x)
```

will print the value 8, and if `n` is equal to 10,

```
subroutine s(y, n)
   use iso_c_binding
   integer(c_int64_t) y(n)
   print *, c_sizeof(y)
end subroutine
```

will print the value 80.

Caution is required when doing mixed-language programming in both C and Fortran, as this is not quite what the C `sizeof` operator does; in many contexts (such as being a dummy argument) a C array 'decays' to a pointer and then `sizeof` will return the size of the pointer (not the whole array) in bytes.

19.8 The value attribute

For the sake of interoperability, there exists an attribute, `value`, for dummy arguments. It may be specified in a type declaration statement for the argument or separately in a `value` statement:

```
function func(a, i, j) bind(c)
   real(c_float) func, a
   integer(c_int), value :: i, j
   value :: a
```

When the procedure is invoked, a copy of the actual argument is made. The dummy argument is a variable that may be altered during execution of the procedure, but on return no copy back takes place. If the type has any length type parameter (character type or a parameterized derived type, see Section 13.2), its value need not be known at compile time. The argument can be an array, but cannot be a pointer, be allocatable, have intent `out` or `inout`, be a procedure, or have the `volatile` attribute (Section 8.11.1).

The `value` attribute is not limited to procedures with the `bind` attribute; it may be used in any procedure. This is useful for a particular programming style; for example, in Figure 19.3, the argument `n` is locally decreased until it reaches zero, without affecting the actual argument or requiring an extra temporary variable. Because the attribute alters the argument-passing mechanism, a procedure with a `value` dummy argument is required to have an explicit interface.

Figure 19.3 Find the *n*th word in a string.

```
integer function nth_word_position(string, n) result(pos)
   character(*), intent(in) :: string
   integer, value           :: n
   logical                  :: in_word
   in_word  = .false.
   do pos = 1, len(string)
      if (string(pos:pos)==' ')then
         in_word = .false.
      else if (.not.in_word) then
         in_word = .true.    ! At first character of a word.
         n = n - 1
         if (n==0) return    ! Found nth one, return position.
      end if
   end do
   pos = 0                   ! n words not found, return zero.
end function
```

In the context of a call from C, the absence of the `value` attribute indicates that it expects the actual argument to be an object pointer to an object of the specified type or a function pointer whose target has a prototype that is interoperable with the specified interface (see next section).

19.9 Interoperability of procedures

A Fortran procedure is interoperable if it has an explicit interface and is declared with the `bind` attribute:

```
function func(i, j, k, l, m) bind(c)
subroutine subr () bind(c)
```

Note that even for a subroutine with no arguments the parentheses are required. The procedure may be an external or module procedure, but is not permitted to be an internal procedure. Each dummy argument must be interoperable and neither optional nor an array with the `value` attribute. For a function, the result must be scalar and interoperable.

The procedure usually has a *binding label*, which has global scope and is the name by which it is known to the C processor.[2] By default, it is the lower-case version of the Fortran name. For example, the function in the previous paragraph has the binding label `func`. An alternative binding label may be specified:

```
function func(i, j, k, l, m) bind(c, name='c_func')
```

The value following the `name=` must be a scalar default character constant expression. Ignoring leading and trailing blanks, this must be a valid C identifier and case is significant.

[2] If an external procedure has a binding label, the procedure name has local scope.

A binding label is not an alias for the procedure name for an ordinary Fortran invocation. It is for use only from C. Two different entities must not have the same binding label.

If the character expression has zero length or is all blanks, there is no binding label. The procedure may still be invoked from C through a procedure pointer and, if this is the only way it will be invoked, it is not appropriate to give it a binding label. In particular, a `private` module procedure must not have a binding label.

An interoperable Fortran procedure interface is interoperable with a C function prototype that has the same number of arguments and does not have variable arguments denoted by the ellipsis (...). For a function, the result must be interoperable with the prototype result. For a subroutine, the prototype must have a void result. A dummy argument with the `value` attribute must be interoperable with the corresponding formal parameter. A dummy argument without the `value` attribute must correspond to a formal parameter of a pointer type and be interoperable with an entity of the referenced type of the formal parameter. Note that a Fortran array is not permitted to have the value attribute, but it can interoperate with a C array since this is automatically of a pointer type.

Here is an example of procedure interface interoperability. The Fortran interface in Figure 19.4 is interoperable with the C function prototype

```
short int func(int i, double *j, int *k, int l[10], void *m);
```

If a C function with this prototype is to be called from Fortran, the Fortran code must access an interface such as this. The call itself is handled in just the same way as if an external Fortran procedure with an explicit interface were being called. This means, for example, that the array section `larray(1:20:2)` might be the actual argument corresponding to the dummy array `l`; in this case, copy-in copy-out takes place.

Figure 19.4 A Fortran interface for a C function.

```
interface
   function func(i, j, k, l, m) bind(c)
   use, intrinsic           :: iso_c_binding
      integer(c_short)      :: func
      integer(c_int), value :: i
      real(c_double)        :: j
      integer(c_int)        :: k, l(10)
      type(c_ptr), value    :: m
   end function func
end interface
```

Similarly, if a Fortran function with the interface of the previous paragraph is to be called from C, the C code must have a prototype such as that of the previous paragraph.

If a C function is called from Fortran, it must not use signal (C standard, 7.14.1) to change the handling of any exception that is being handled by the Fortran processor, and it must not alter the floating-point status (Section 18.8.4) other than by setting an exception flag to signaling. The values of the floating-point exception flags on entry to a C function are processor dependent.

19.10 Interoperability of global data

An interoperable module variable (or a common block, Appendix B.3.3, with interoperable members) may be given the `bind` attribute in a type declaration statement or in a `bind` statement:

```
use iso_c_binding
integer(c_int), bind(c) :: c_extern
integer(c_long) :: c2
bind(c, name='myvariable') :: c2
common /com/ r, s
real(c_float) :: r, s
bind(c) :: /com/
```

It has a binding label defined by the same rules as for procedures, and interoperates with a C variable with external linkage that is of a corresponding type. If a binding label is specified in a statement, the statement must define a single variable.

A variable with the `bind` attribute also has the `save` attribute (which may be confirmed explicitly). A change to the variable in either language affects the value of the corresponding variable in the other language. This is known as **linkage association**. A C variable is not permitted to interoperate with more than one Fortran variable.

The `bind` statement is available only for this purpose; it is not available, for instance, to specify the `bind` attribute for a module procedure. Also, the `bind` attribute must not be specified for a variable that is not a module variable (that is, it is not available to confirm that a variable is interoperable), and it must not be specified for a module variable that is in a common block.

If a common block is specified in a `bind` statement, it must be specified in a `bind` statement with the same binding label in every scoping unit in which it is declared. It interoperates with a variable of structure type whose components are each interoperable with the corresponding member of the common block. If the common block has only one member, it also interoperates with a variable that is interoperable with the member.

The `equivalence` statement (Appendix B.3.2) is not permitted to specify a variable that has the `bind` attribute or is a member of a `common` block that has the `bind` attribute.

The double colon in a `bind` statement is optional.

19.11 Invoking a C function from Fortran

If a C function is to be invoked from Fortran, it must have external linkage and be describable by a C prototype that is interoperable with an accessible Fortran interface that has the same binding label.

If it is required to pass a Fortran array to C, the interface may specify the array to be of explicit or assumed size and the usual Fortran mechanisms, perhaps involving copy-in copy-out, ensure that a contiguous array is received by the C code. Here is an example involving both an assumed-size array and an allocatable array. The C prototype is

```
int c_library_function(int expl[100], float alloc[], int len_alloc);
```

Figure 19.5 Passing Fortran arrays to a C function.

```
use iso_c_binding
interface
   integer (c_int) function c_library_function         &
                   (expl, alloc, len_alloc) bind(c)
      use iso_c_binding
      integer(c_int)        :: expl(100)
      real(c_float)         :: alloc(*)
      integer(c_int), value :: len_alloc
   end function c_library_function
end interface
integer(c_int)             :: expl(100), len_alloc, x1
real(c_float), allocatable :: alloc(:)
⋮
len_alloc = 200
allocate (alloc(len_alloc))
⋮
x1 = c_library_function(expl, alloc, len_alloc)
⋮
```

and the Fortran code is shown in Figure 19.5.

The rules on shape and character length disagreement (Section 19.5) allow entities specified as `character(kind=c_char)` of any length to be associated with an assumed-size or explicit-shape array, and thus to be passed to and from C. For example, the C function with prototype

```
void Copy(char in[], char out[]);
```

may be invoked by the Fortran code in Figure 19.6.

This code works because Fortran allows the character variable `digit_string` to be associated with the assumed-size dummy array `in`. We have also taken the opportunity here to illustrate the use of a binding label to call a C procedure whose name includes an upper-case letter.

19.12 Invoking Fortran from C

A reference in C to a procedure that has the `bind` attribute, has the same binding label, and is defined by means of Fortran, causes the Fortran procedure to be invoked.

Figure 19.7 shows an example of a Fortran procedure that is called from C and uses a structure to enable arrays allocated in C to be accessed in Fortran. The corresponding C structure declaration is:

```
structure pass { int lenc, lenf;  float *c, *f; };
```

the C function prototype is:

Figure 19.6 Passing Fortran character strings to a C function.

```
use, intrinsic :: iso_c_binding, only: c_char, c_null_char
interface
   subroutine copy(in, out) bind(c, name='Copy')
      use, intrinsic :: iso_c_binding, only: c_char
      character(kind=c_char), dimension(*) :: in, out
   end subroutine copy
end interface
character(len=10, kind=c_char) :: &
             digit_string = c_char_'123456789' // c_null_char
character(kind=c_char) :: digit_arr(10)
call copy(digit_string, digit_arr)
print '(1x, a1)', digit_arr(1:9)
end
```

```
void simulation(struct pass *arrays);
```

and the C calling statement might be:

```
simulation(&arrays);
```

Figure 19.7 Accessing in Fortran an array that was allocated in C.

```
subroutine simulation(arrays) bind(c)
   use iso_c_binding
   type, bind(c) :: pass
      integer (c_int) :: lenc, lenf
      type (c_ptr)    :: c, f
   end type pass
   type (pass), intent(in) :: arrays
   real (c_float), pointer :: c_array(:)
   ⋮
   ! associate c_array with an array allocated in C
   call c_f_pointer(arrays%c, c_array, (/arrays%lenc/) )
   ⋮
end subroutine simulation
```

It is not uncommon for a Fortran library module to have an initialization procedure that establishes a data structure to hold all the data for a particular problem that is to be solved. Subsequent calls to other procedures in the module provide data about the problem or receive data about its solution. The data structure is likely to be of a type that is not interoperable, for example because it has components that are allocatable arrays.

The procedures `c_loc` and `c_f_pointer` have been designed to support this situation. The Fortran code in Figure 19.8 illustrates this. The type `problem_struct` holds an allocatable

array of the size of the problem, and lots more. When the C code calls `new_problem`, it passes the size. The Fortran code allocates a structure and an array component within it of the relevant size; it then returns a pointer to the structure. The C code later calls `add` and passes additional data together with the pointer that it received from `new_problem`. The Fortran procedure `add` uses `c_f_pointer` to establish a Fortran pointer for the relevant structure and performs calculations using it. Note that the C code may call `new_problem` several times if it wishes to work simultaneously with several problems; each will have a separate structure of type `problem_struct` and be accessible through its own 'handle' of `type(c_ptr)`. When a problem is complete, the C code calls `goodbye` to deallocate its structure.

Figure 19.8 Providing access in C to a Fortran structure that is not interoperable.

```
module lib_code
   use iso_c_binding
   type :: problem_struct
      real, allocatable :: a(:)
      ⋮ ! More stuff
   end type
contains
   type(c_ptr) function new_problem(problem_size) bind(c)
      integer(c_size_t), value      :: problem_size
      type(problem_struct), pointer :: problem_ptr
      allocate(problem_ptr)
      allocate(problem_ptr%a(problem_size))
      new_problem = c_loc(problem_ptr)
   end function new_problem
   subroutine add(problem,...) bind(c)
      type(c_ptr), intent(in), value :: problem
      type(problem_struct), pointer  :: problem_ptr
      ⋮
      call c_f_pointer(problem, problem_ptr)
      ⋮
   end subroutine add
   subroutine goodbye(problem) bind(c)
      type(c_ptr), intent(in), value :: problem
      type(problem_struct), pointer  :: problem_ptr
      call c_f_pointer(problem, problem_ptr)
      deallocate(problem_ptr)
   end subroutine goodbye
end module lib_code
```

19.13 Enumerations

An enumeration is a set of integer constants (enumerators) that is appropriate for interoperating with C. The kind of the enumerators corresponds to the integer type that C would choose for the same set of constants. Here is an example:

```
enum, bind(c)
   enumerator :: red = 4, blue = 9
   enumerator yellow
end enum
```

This declares the named constants `red`, `blue`, and `yellow` with values 4, 9, and 10, respectively.

If a value is not specified for an enumerator, it is taken as one greater than the previous enumerator or zero if it is the first.

To declare a variable of the enumeration type, use the `kind` intrinsic function on one of the constants. An example using the above `enum` definition is:

```
integer(kind(red)) :: background_colour
```

Exercises

1. Write a generic Fortran interface block for the standard C libm error functions `erf` and `erff`.

2. Write Fortran functions to compute the dot product of two vectors, suitable for being called from C.

20. Fortran 2018 coarray enhancements

20.1 Teams

Teams have been introduced to allow separate sets of images to execute independently. An important design objective was that, given code that has been developed and tested on all images, it should be possible to run the code on a team without making changes. This requires that if a team has *n* images, the image indices within the team run from 1 to *n*.

It was decided that teams should always be formed by partitioning an existing team into parts, starting with the team of all the images, which is known as the **initial team**. The team in which a statement is executed by an image is known as the **current team**.

Information about a team is held collectively on all the images of the team in a scalar variable of type `team_type` from the intrinsic module `iso_fortran_env`. The components of this type are private, and may contain information to improve communication efficiency within the team. To facilitate implementation where this information may be different on different images, a team variable created on one image is not usable on another image. Therefore, a coarray or coindexed object must not be of type `team_type`, and a polymorphic coarray must not be allocated to be of type `team_type` or to have a subobject of type `team_type`. Furthermore, assigning a value of type `team_type` to a variable on another image, or vice versa, causes the variable to become undefined. (These restrictions and effects are the same as for type `c_ptr`.)

A set of new teams is formed by executing a `form team` statement on all images of the current team. The new team to which an image of the current team will belong is determined by its **team number**, which is a positive integer. All the images with the same team number will belong to the same new team. The team number is specified on the `form team` statement by an integer expression. For example, the code

```
use iso_fortran_env
type ( team_type ) new_team
⋮
form team ( 1 + mod(this_image(),2), new_team )
```

forms two new teams consisting of the images of the current team that have odd or even image indices. We describe the `form team` statement in detail in Section 20.3.

Changes of team take place at the `change team` and `end team` statements, which mark the beginning and end of a new construct, the `change team` construct:

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

```
change team ( new_team )
    ⋮ ! Statements executed with new_team as the current team
end team
```

The values of `new_team` must have been established by the prior execution of a `form team` statement on all the images of the current team. Both the `change team` and the `end team` statement are image control statements. The executing image and the other images of its new team synchronize at these statements. These images must all execute the same `change team` statement. While we expect it to be usual for the images of the other new teams to execute the construct, this is not required – they might continue to execute in the previously current team. We describe the `change team` construct in detail in Section 20.4.

The team that is current during the execution of a `form team` statement is known as the **parent** of each new team formed. We find it convenient to refer to each new team as a **child** of the current team. If the current team is not the initial team, the children have as parent a team that itself has a parent. Parents of parents can occur at any depth and are known as **ancestors**. Note that `change team` constructs can be nested to any depth and can be executed in a procedure called from within a `change team` construct.

20.2 Image failure

It is anticipated that a coarray program might execute on a huge number of images. While the likelihood of a particular image failing during the execution of a program is small, the likelihood that one of them might fail is significant when there are a huge number of them. Therefore, the concept of continued execution in the presence of failed images has been introduced. It is not required that the system support this feature, and even if it does, not all failure scenarios are covered by this model.

The named constant `stat_failed_image` has been added to the intrinsic module `iso_fortran_env`. This is positive if failed image handling is supported and negative otherwise. If it is positive, it is used for the value of a `stat=` specifier or `stat` argument if a failed image is involved in either an image control statement, a reference to a coindexed object (see Section 20.9), or an invocation of a collective (Section 20.19) or atomic (Section 20.20) subroutine, and no other error condition occurs.

The system will not automatically produce correct results in the presence of failed images. Instead, these language features are designed to enable the program to detect failure, and initiate a recovery process. For example, it may be possible to go back to a previous state of the computation and repeat it with fewer images, or with 'reserve' images brought in to replace failed ones. There are likely to be only a few key points in the computation from which recovery is practical, so it is important that the working images all reach such a point, albeit with incorrect results. For example, the `change team` construct does not fail in the presence of failed images – instead, it executes on all the remaining images of the team. To allow such continued execution, `stat=` specifiers (see Section 20.21) have been added to image selectors and additional image control statements. The term **active** has been introduced for an image that has neither failed nor stopped.

20.3 Form team statement

The `form team` statement takes the general form

`form team (` *team-number*`,` *team-variable* `[ ,` *form-team-spec* `] ... )`

where *team-number* is a scalar integer expression whose value must be positive, *team-variable* is a scalar variable of type `team_type`, and each *form-team-spec* is a `new_index=`, `stat=`, or `errmsg=` specifier. The *team-variable* on each image of a child team will be defined with information about that child team. All the images of the current team that specify the same *team-number* value will be in the same child team. This value is the **team number** for the child team and can be used to identify it in statements executed on an image of another child of the same parent. The team number of any image in the initial team is equal to -1.

By default, the processor chooses which image indices are assigned to which images of each child team, and the choice may vary from processor to processor. However, they may be specified by including the *form-team-spec*

`new_index=`*expr*

in the statement, where *expr* is a scalar integer expression; this specifier provides the image index for the executing image in the child team. If a child team has k images, the values on those images must be a permutation of $1, 2, \ldots, k$.

The `form team` statement is an image control statement. The same statement must be executed by all active images of the current team, and they synchronize.

20.4 Change team construct

The `change team` statement takes the general form

[construct-name `:` *]* `change team (` *team-value [,* *association* *] ... [,* *sync-stat-list* *]* `)`

where *team-value* is a scalar of type `team_type`. Its values on the images that execute the same `change team` statement must be as constructed by corresponding executions of a `form team` statement on those images (see Section 20.1). If it is a variable, its value must not be altered during the execution of the `change team` construct.

Each *association* has the form

coarray-name `[` *coarray-spec* `]` `=>` *selector*

and declares a new name and new cobounds for *selector*, which is a named **associating** coarray that is in scope at the `change team` statement. For example, if `big` has cobounds `[1:k,1:k]` and the current team is subdivided into k teams of k images, the *association*

```
part[*] => big
```

makes `part` an associating coarray with cobounds `[1:k]` on each new team. The appearance of an *association* does not prevent the coarray being referenced by its original name and with its original cosubscripts. The mapping to an image index is unchanged but the image index will refer to the image of the current (new) team with that index. The largest value that the final cobound can have in a valid reference is returned by the intrinsic `ucobound`.

The *sync-stat-list* is as for the `sync all` statement (Section 17.5). If a child image has failed but no other error occurs, `stat_failed_image` is assigned to the `stat=` variable and all the active images of the child team synchronize on entering the construct, the same as if the execution were successful.

The `end team` statement takes the general form

`end team` *[* `(` *[sync-stat-list]* `)` *] [construct-name]*

The reason for the appearance of *sync-stat-list* here is to detect the possibility of image failure in the current team during execution of the construct. The `end team` statement is an image control statement and the images of the team that were current inside the construct synchronize here. After the execution of `end team` completes, the current team reverts to the one current before the matching `change team` statement execution.

20.5 Coarrays allocated in teams

In Fortran 2008, coarrays are always allocated and deallocated in synchrony across all images, which allows each image to calculate the address of a coarray element on another image (symmetric memory). In Fortran 2018, synchronization is now across the team, of course. Symmetric memory is maintained within teams by requiring that

- any allocatable coarray that is allocated before entry to a `change team` construct remains allocated during the execution of the construct, and
- any allocatable coarray that becomes allocated within a `change team` construct and is still allocated when the construct is left is automatically deallocated, even if it has the `save` attribute.

This allows each image to hold its allocatable coarrays in a stack with those allocated in the initial team at the bottom, those allocated in the team that is the child of the initial team next, those allocated in the team that is the grandchild of the initial team next, etc. Of course, there is no requirement for exactly this form of memory management to be used.

20.6 Critical construct and image failure

The `critical` statement now takes the general form

[construct-name `:` *]* `critical` *[* `(` *[sync-stat-list]* `)` *]*

with optional `stat=` and `errmsg=` specifiers in its *sync-stat-list* to detect the case of an image failing while executing a `critical` construct. If `stat=` is present in this case, the construct is treated as having completed execution so that another image can commence executing it. When this other image commences execution of the construct, the `stat=` variable will have the value `stat_failed_image` to indicate the failure of the previous execution of the construct. The `errmsg=` specifier provides a message in the event of failure.

20.7 Lock and unlock statements and image failure

Failure of an image causes all lock variables that are locked by that image to become unlocked.

If a `stat=` specifier is present in a `lock` statement and the image of the lock variable has failed, the `stat=` variable is given the value `stat_failed_image`. Otherwise, if the lock variable is unlocked because of the failure of the image that locked it, the `stat=` variable

is given the value `stat_unlocked_failed_image` from the module `iso_fortran_env`. If a `stat=` specifier is present in an `unlock` statement and the image of the lock variable has failed, the stat variable is given the value `stat_failed_image`.

20.8 Sync team statement

The `sync team` statement has been introduced to allow synchronization within an ancestor team without leaving a `change team` construct, or within a child team to which the executing image belongs without entering a `change team` construct. It has the general form

`sync team (` *[team-value [, sync-stat-list]* `)`

where *team-value* is of type `team_type` and identifies a child team, the current team, or an ancestor team. Successful execution synchronizes all the images of the specified team in the same way as `sync all` does for the current team.

The *sync-stat-list* is as for the `sync all` statement.

20.9 Image selectors

In Fortran 2008, an image is selected in a coindexed object by a list of cosubscripts in square brackets. To allow for failed images and for wishing to address an image in another team, this has been generalized to the **image selector** with the general form

`[` *cosubscript-list [, is-stat-list]* `]`

where *is-stat* is `stat=`*stat-variable*, `team=`*team-value*, or `team_number=`*expr*; *stat-variable* is a scalar non-coindexed integer variable that should be able to represent $-9999 \ldots +9999$, *team-value* is a scalar expression of type `team_type`, and *expr* is a scalar integer expression.

An image selector with a `stat=` specifier permits the programmer to detect the case where the image selected has failed, in which case the *stat-variable* is given the value `stat_failed_image`. Otherwise, it is given the value zero. Execution always continues if the image has failed, whether or not there is a `stat=` specifier. The value obtained on a reference is processor dependent. For a definition, there is no effect except for defining the *stat-variable* if it appears.

Consider a coarray `old` that is in scope at a `change team` statement. If this is accessed within the `change team` construct using its name, cosubscripts map to an image index just as they do outside the construct, but this refers to an image within the current team. However, it is likely that data from other teams will need to be accessed from time to time, subject to suitable synchronization. Significant overheads are likely to be associated with leaving the construct, performing the data exchange, and changing teams again. Instead, an image of a sibling team may be accessed thus

`old[` *cosubscript-list*`, team_number=`*expr* `]`

where the value of *expr* is equal to a team number of any team formed by the execution of the `form team` statement that created the current team. The coarray must be established (Section 20.10) in an ancestor of the current team. An image of an ancestor team (or the current team) may be accessed thus

`old[` *cosubscript-list*`, team=`*team-value* `]`

The `team=` and `team_number=` specifiers are mutually exclusive, but the `stat=` specifier may appear with or without either of them.

20.10 Procedure calls and teams

When a procedure with a coarray dummy argument is called, the current team does not change but its siblings and ancestors are not available to the dummy coarray in image selectors. Indeed, it is hard to see how the compiled code could cope with different sets of nested `change team` constructs at the points of invocation. Because of this, the concept of 'establishment' for a coarray in a team has been introduced.

A non-allocatable coarray with the `save` attribute is **established** in the initial team. An allocated allocatable coarray is **established** in the team in which it was allocated. An unallocated allocatable coarray is not established. An associating coarray in a `change team` construct (Section 20.4) is **established** in the team that is current in the `change team` construct. A non-allocatable coarray that is an associating entity in an `associate`, `select rank`, or `select type` construct is **established** in the team in which the `associate`, `select rank`, or `select type` statement is executed. A non-allocatable coarray that is a dummy argument or host associated with a dummy argument is **established** in the team in which the procedure was invoked. A coarray dummy argument is not established in any ancestor team.

20.11 Intrinsic functions get_team and team_number

`get_team ( [ level ] )` returns a value of type `team_type`; `level` is an optional integer scalar with value one of the constants `initial_team`, `parent_team`, and `current_team` in the intrinsic module `iso_fortran_env`. The function returns a team value that identifies the current team if `level` is not present or the team indicated by `level` if it is present.

`team_number ( [ team ] )` returns a value of type default integer. It is the team number (Section 20.3) of the specified team within its parent team; `team` is an optional scalar value of type `team_type` that specifies the current or an ancestor team. Absence specifies the current team.

Referencing `get_team` is the only way to obtain the team value of the initial team; this will be needed in order to refer to it in a `sync_team` statement or image selector when executing in a child team.

Referencing `team_number` allows the executing image to determine in which team it lies and execute appropriate code, as illustrated in Figure 20.1.

20.12 Intrinsic function image_index

The intrinsic function `image_index` now has two additional forms:

`image_index (coarray, sub, team)` returns a default integer scalar. If `sub` holds a valid sequence of cosubscripts for `coarray` in the team `team`, the result is the corresponding image index. Otherwise, the result is zero.

Figure 20.1 Use of `team_number`.

```
change team (odd_even)
   select case (team_number())
      case (1)
         ⋮  ! Code for images in team 1.
      case (2)
         ⋮  ! Code for images in team 2.
   end select
   ⋮
end team
```

`coarray` is a coarray that can have cosubscripts and is of any type.
`sub` is a rank-one integer array of size equal to the corank of `coarray` in the team `team`.
`team` is a scalar value of type `team_type` that specifies the current or an ancestor team.

`image_index (coarray, sub, team_number)` returns a default integer scalar. If `sub` holds a valid sequence of cosubscripts for `coarray` in the team `team_number`, the result is the corresponding image index. Otherwise, the result is zero.
`coarray` is a coarray that can have cosubscripts and is of any type.
`sub` is a rank-one integer array of size equal to the corank of `coarray` in the sibling team specified by `team_number`.
`team_number` is an integer scalar value identifying a sibling team of the current team.

For a given set of cosubscripts, the value of the function may change on entering a `change team` construct. For example, `image_index (coarray, [30])` might be 30 in the parent team and 0 in the child team if this has only 15 images. Therefore, `image_index` cannot remain an inquiry function and becomes transformational.

20.13 Intrinsic function num_images

The intrinsic function `num_images` now has two additional forms:

`num_images (team)` returns the number of images in the team `team` as a default integer scalar.
`team` is a scalar value of type `team_type` that identifies the current or an ancestor team.

`num_images (team_number)` returns the number of images in the sibling team identified by `team_number` as a default integer scalar.
`team_number` is an integer scalar value identifying a sibling team of the current team.

20.14 Intrinsic function this_image

The intrinsic function `this_image` now has three forms:

`this_image ([team])` returns the image index of the executing image in the team `team`, or the current team if `team` is absent, as a default integer scalar.

`this_image (coarray[, team])` returns a default integer rank-one array holding the sequence of cosubscript values for `coarray` that would specify the executing image in the team specified by `team`, or the current team if `team` is absent.

`this_image (coarray, dim[, team])` returns the value of cosubscript `dim` in the sequence of cosubscript values for `coarray` that would specify the executing image in the team specified by `team`, or the current team if `team` is absent, as a default integer scalar.

`coarray` is a coarray of any type. If allocatable, it must be allocated. If it is of type `team_type`, the argument `team` must be present.

`dim` is an integer scalar value in the range $1 \leq$ `dim` $\leq n$ where n is the corank of `coarray` in the specified team.

`team` is a scalar value of type `team_type` that identifies the current or an ancestor team.

20.15 Intrinsic function coshape

The intrinsic function `coshape` has been added for consistency with `lcobound` and `ucobound`.

`coshape(coarray[, kind])` returns a rank-one integer array whose size is the corank of `coarray`. It is of kind `kind` if it is present, and default kind otherwise. Element i has the value $1 +$ `ucobound(coarray,`i`)` $-$ `lcobound(coarray,`i`)`.

`coarray` is a coarray that is permitted to have subscripts. It may be of any type. If it is allocatable, it must be allocated.

`kind` is a scalar integer constant expression.

20.16 Intrinsic function move_alloc

The intrinsic function `move_alloc` has had additional optional arguments `stat` and `errmsg` added and now has the form:

`call move_alloc( from, to[, stat][, errmsg])`

`stat` is an optional integer scalar with a decimal exponent range of at least 4. It has intent `out`. It must not be coindexed. If it is not present and an error occurs, error termination is initiated. If the execution is successful, `stat` is assigned the value zero. If an error occurs, it is assigned a nonzero value as follows:

- if `from` and `to` are coarrays, and the current team contains a stopped image, it is assigned the value `stat_stopped_image`;
- otherwise, if `from` and `to` are coarrays, the current team contains a failed image, and no other error occurs, it is assigned the value `stat_failed_image`;
- otherwise, it is assigned a positive value that differs from the values of `stat_failed_image` and `stat_stopped_image`.

errmsg is an optional intent `inout` scalar of type default character. It must not be coindexed. When present, it provides an explanatory message in the event of an error condition, and is unchanged otherwise.

Successful execution is as in Fortran 2008 except that it applies to the current team. If `stat` is given the value `stat_failed_image`, the coarrays `from` and `to` will have had their allocation moved on the active images.

20.17 Fail image statement

The statement

```
fail image
```

causes the executing image to behave as if it has failed. No further statements are executed by the image. Its purpose is to facilitate testing of image failure recovery methods.

20.18 Detecting failed and stopped images

Three new intrinsic functions have been added to assist the detection of failed and stopped images. There are two transformational functions:

failed_images (*[*team*][*, kind*]*)
stopped_images (*[*team*][*, kind*]*)

These functions return a rank-one integer array holding the image indices of the images in the specified team that are known to have failed or stopped respectively, in numerically increasing order.

team is an optional scalar value of type `team_type` that identifies the current or an ancestor team; absence specifies the current team.

kind is an optional integer scalar constant expression that specifies the kind of the result; if absent, the result is of default kind.

The elemental function

image_status (image*[*, team*]*) returns a value of type default integer that is `stat_failed_image` if the image with index `image` has failed, `stat_stopped_image` if it has stopped, and zero otherwise.

image is a scalar integer whose value must be the image index of an image in the specified team.

team is an optional scalar value of type `team_type` that identifies the current or an ancestor team; absence specifies the current team.

20.19 Collective subroutines

Intrinsic subroutines have been added to perform collective operations on all the images of a team, such as summing the values of a variable across the images. It is to be expected that the execution will have been optimized by the system, for example by associating the images of the team with the leaves of a binary tree and grouping the operations by tree level.

The same `call` statement must be executed on all active images of the team and it must occur in a context that would allow an image control statement. There is no automatic synchronization at the statement, but it is to be expected that the system applies some form of synchronization while executing the subroutine. To avoid the possibility of these synchronizations causing deadlock, the sequence of invocations must be the same on all active images of the team from the beginning to the end of execution as a team.

All or almost all of the subroutines have these arguments:

a is a scalar or an array that has the same shape on all the images of the current team. It has intent `inout`. It must not be coindexed. It may be a coarray but this is not required.

result_image is an optional intent `in` integer scalar. If present, it must have the same value on all images of the team. It specifies the image on which the result is placed and `a` becomes undefined on all other images of the team. If it is not present, the result is broadcast to all images of the team.

stat is an optional intent `out` integer scalar. If it is not present and an error occurs, error termination is initiated. If present on one image, it must be present on all images of the team. It must not be coindexed. If the invocation is successful, it is given the value zero. If an error occurs, it is given a nonzero value and the argument `a` becomes undefined. If there is a stopped image, it is given the value `stat_stopped_image`. Otherwise, if there is a failed image, it is given the value `stat_failed_image`. Otherwise, it is given another value.

errmsg is an optional intent `inout` scalar of type default character. It must not be coindexed. When present, it provides an explanatory message in the event of an error condition.

The subroutines are as follows:

call co_broadcast(a, source_image*[*, stat*][*, errmsg*]) copies the value of `a` on `source_image` to the corresponding argument on all the other images of the team as if by intrinsic assignment. The variable `a` may have any type, but its dynamic type and type parameter values must be the same on all images of the team.

source_image is an intent `in` integer scalar that specifies the image from which values are broadcast. It must have the same value on all images of the team.

call co_max(a[, result_image][, stat][, errmsg]) computes the maximum value of `a` on all images of the team. The result is placed in `a` on `result_image` if this is present and otherwise in `a` on all images of the team. The variable `a` may be of type integer, real, or character, but must have the same type and type parameters on all images of the team. If it is an array, the result is computed element by element.

call co_min(a[, result_image][, stat][, errmsg]) behaves like the subroutine `co_max` but is for minimum instead of maximum values.

call co_sum(a[, result_image][, stat][, errmsg]) computes the sum of the values of `a` on all images of the team. The variable `a` may be of any numeric type, but must have the same type and type parameters on all images of the team. If it is an array, the result is computed element by element. The order of summation is not specified so the result may be affected by rounding errors that can vary even between different runs on the same computer. The result is placed in `a` on `result_image` if this is present and otherwise exactly the same result is placed in `a` on all images of the team.

call co_reduce(a, operation[, result_image][, stat][,errmsg]) behaves like `co_sum` but reduces the coarray by a supplied function instead of by summation. The variable `a` may be of any type but must not be polymorphic.

operation is a pure function with two scalar dummy arguments and a scalar result, all of the type and type parameters of `a`. Neither argument may be polymorphic, allocatable, optional, or a pointer. If one has the `asynchronous`, `target`, or `value` attribute, the other must too. It must be the same function on all images of the team and must implement an operation that is mathematically associative but need not be computationally associative (for example, floating-point addition is not computationally associative due to rounding).

20.20 New atomic subroutines

Nine atomic subroutines have been added. All or many of them have these arguments:

atom is a scalar coarray or coindexed object of type `integer(atomic_int_kind)` or, alternatively, for `atomic_cas` only, `logical(atomic_logical_kind)`. It has intent `inout`.

value is an integer scalar. It has intent `in`.

old is a scalar of the same type and kind as `atom`. It has intent `out`.

stat is an optional integer scalar with a decimal exponent range of at least 4. It must not be coindexed. It has intent `out`. If it is not present and an error occurs, error termination is initiated. If the invocation is successful, it is given the value zero. If an error occurs,

it is given a nonzero value and any `atom` or `old` arguments become undefined. If `atom` is on a failed image and there is no other cause for the error, the value given is `stat_failed_image`.

If the `stat` argument is present and no error condition occurs, it is given the value 0. If an error condition occurs, any `atom` or `old` argument becomes undefined.

The subroutines are as follows:

`call atomic_add ( atom, value[ , stat ])` gives the variable `atom` the value `atom+int(value,atomic_int_kind)`.

`call atomic_and ( atom, value[ , stat ])` gives the variable `atom` the value `iand(atom,int(value,atomic_int_kind))`.

`call atomic_or ( atom, value[ , stat ])` gives the variable `atom` the value `ior(atom,int(value,atomic_int_kind))`.

`call atomic_xor ( atom, value[ , stat ])` gives the variable `atom` the value `ieor(atom,int(value,atomic_int_kind))`.

`call atomic_fetch_add ( atom, value, old[ , stat ])` gives the variable `atom` the value `atom+int(value,atomic_int_kind)` and `old` is given the value `atom` had on entry.

`call atomic_fetch_and ( atom, value, old[ , stat ])` gives the variable `atom` the value `iand(atom,int(value,atomic_int_kind))` and `old` is given the value `atom` had on entry.

`call atomic_fetch_or ( atom, value, old[ , stat ])` gives the variable `atom` the value `ior(atom,int(value,atomic_int_kind))` and `old` is given the value `atom` had on entry.

`call atomic_fetch_xor ( atom, value, old[ , stat ])` gives the variable `atom` the value `ieor(atom,int(value,atomic_int_kind))` and `old` is given the value `atom` had on entry.

`call atomic_cas ( atom, old, compare, new[ , stat ])` compares the value of `atom` with that of `compare`. If they are equal, it causes `atom` to be given the value of `int(new,atomic_int_kind)` if it is of type integer and `new` if it is of type logical. Otherwise, the value of `atom` is not changed. In either case, `old` is given the value `atom` had on entry.

`compare` is a scalar of the same type and kind as `atom`. It has intent `in`.

`new` is a scalar of the same type and kind as `atom`. It has intent `in`.

20.21 Failed images and stat= specifiers

If a `stat=` specifier is present in a `change team`, `end team`, `event post`, `form team`, `sync all`, `sync images`, or `sync team` statement, and one of the images involved has failed but none has stopped, and no other error condition occurs, the variable in the `stat=` specifier is assigned the value `stat_failed_image`, and the intended action takes place on the active images involved as if the execution were successful.

20.22 Events

Events have been introduced to allow an action to be delayed until one or more actions have been performed on other images. An image records that it has performed an action by executing an `event post` statement involving a scalar variable of type `event_type` from the intrinsic module `iso_fortran_env`, known as an **event variable**. An event variable must be a coarray or a component of a coarray; when posting an event, it is almost always coindexed, but only the owning image can wait on an event.

An image executes an `event wait` statement involving the event variable if it needs to delay its action for a posted action on another image or for $k \geq 1$ posted actions on other images. This event variable is not allowed to be a coindexed object. The action on the waiting image is said to be **matched** by each of the k posted actions. The count of the number of posted actions that have not yet been matched is held in the event variable. This is known as the **event count**.

Each `event post` execution has a matching `event wait` execution that involves the same event variable. A segment that precedes the `event post` execution precedes any segment that succeeds the matching `event wait` execution.

The value of an event variable includes its event count, which is of type integer with kind `atomic_int_kind`, and intially has the value zero. It is atomically incremented by one when an `event post` statement is executed for it and is atomically decremented by the chosen threshold k when an `event wait` statement is executed for it.

The type `event_type` is extensible and has no type parameters. All its components are private. An event variable may be defined only by appearance in an `event post` or `event wait` statement. It may be referenced or defined in a segment that is unordered with respect to another segment in which it is defined.

The `event post` statement is an image control statement that takes the general form

`event post (` *event-variable* `[ ,` *sync-stat-list* `] )`

where *sync-stat-list* is as for the `sync all` statement. Successful completion of the statement atomically increases its event count by one. If there is an error condition, the value of the event count is processor dependent.

The `event wait` statement is an image control statement that takes the general form

`event wait (` *event-variable* `[ ,` *event-wait-spec-list* `] )`

where an *event-wait-spec* is an `until_count=` specifier, `stat=` specifier, or `errmsg=` specifier. The event variable must not be a coindexed object. An `until_count=` specifier has the form

`until_count =` *scalar-integer-expression*

where the value of the expression provides the threshold. The threshold is 1 if there is no `until_count=` specifier. The executing image waits until the event count is at or above the threshold, then atomically decreases the count by the threshold and resumes execution. If the threshold is k, the first k unmatched `event post` executions for the event variable are matched with this `event wait` execution. After an error condition, the value of the event count is processor dependent.

Note that if an error occurs in an `event post` or `event wait` statement, error termination will be initiated.

The value of an event count may be determined by the intrinsic subroutine

`call event_query( event, count[, stat])`

`event` is an event variable that is not coindexed. It has intent `in`.

`count` is a scalar integer with decimal range at least that of default integer. It has intent `out`. It is assigned the value of the count of `event`. Note that `event` is accessed atomically, but may have already changed by the time `event_query` returns.

`stat` is scalar integer with decimal range of at least four. It must not be a coindexed object. If present, it is assigned the value zero if no error occurs, and a positive value otherwise. If an error occurs and `stat` is absent, error termination is initiated.

21. Fortran 2018 enhancements to interoperability with C

21.1 Introduction

The design of the C interoperability features in Fortran 2003 followed the principle that interoperability was only possible where both C and Fortran had very similar features. Although this led to semantics that are relatively easy to understand, and to write interoperable procedures for, it has meant that some very useful Fortran features are not available when interoperating with C; in particular,

- optional dummy arguments,
- assumed-length character dummy arguments,
- assumed-shape arrays,
- allocatable dummy arguments, and
- pointer dummy arguments.

Also, the frequent need to pass C pointers for low-level operations (using `c_loc`) can lead to ugly code that is difficult to understand.

Fortran 2018 addresses all of these issues. For optional arguments there are already widespread C programming idioms that can be followed, and so this was the simplest deficiency to correct. For all the other advanced Fortran features, the necessary information that needs to be passed to/from the C functions is passed with a **C descriptor**, and there are mechanisms provided for the C functions to use such descriptors. For more convenient low-level C interoperability, **assumed type** dummy arguments are available.

Additionally, the new **assumed rank** feature can also be used in interoperable procedures.

We will now discuss the simplest new C interoperability feature, optional dummy argument handling.

21.2 Optional arguments

The C programming language does not have any direct equivalent of Fortran optional arguments, and so Fortran 2003 did not provide for any interoperability with them. However, a widespread programming idiom in C with a similar effect to optional arguments is to pass the argument by reference, with a null pointer to indicate that the argument is not present.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

For example, the second argument of the standard C library function `strtod` is permitted to be a null pointer; if it is not a null pointer, it is assigned a value through the pointer.

Fortran 2018 adopts this idiom for permitting Fortran interoperable procedures to have optional arguments. As the idiom needs the argument to be passed by reference, an optional argument of an interoperable procedure is not permitted to have the `value` attribute.

For example, the interface

```
function strtod(string, final)
   use iso_c_binding
   character(kind=c_char) string(*)
   type(c_ptr),optional :: final
   real(c_double) strtod
end function strtod
```

allows calling the standard C library function `strtod` directly, without needing to explicitly code the passing of a null pointer, or the address of the second argument. For example,

```
real(c_double) x, y
type(c_ptr) yfinal
⋮
x = strtod(xstring//c_null_char)
y = strtod(ystring//c_null_char, yfinal)
```

In a call from Fortran, when the actual argument is absent, a null pointer will be passed to the C procedure. Similarly, if the interoperable procedure is written in Fortran, when called from C with a null pointer for the optional argument it will be treated as absent.

For example, for the Fortran procedure

```
subroutine fortran_note(main, minor) bind(c)
   use iso_c_binding
   integer(c_int), value :: main
   integer(c_int), optional :: minor
   if (present(minor)) then
      print '(1X,"Note ",I0,".",I0)', main, minor
   else
      print '(1X,"Note ",I0)', main
   end if
end subroutine fortran_note
```

the invocation from C

```
fortran_note(10, (int *)0);
```

would print '`Note 10`', whereas the invocation

```
int subnote = 3;
fortran_stop(11,&subnote);
```

would print '`Note 10.3`'.

21.3 Low-level C interoperability

Although Fortran 2008 provides interoperation with C '`void *`' arguments via type `c_ptr`, this can be inconvenient when calling generic C functions that operate on any type, for example as a simple blob of memory. This is addressed in Fortran 2018 with **assumed-type** dummy arguments that are either scalar or assumed size (Section 19.5).

For example, a C function that produces a 64-bit checksum of a block of memory of arbitrary size might have the signature

```
extern uint64_t checksum(void *block,size_t nbytes);
```

and this could be used in Fortran as follows:

```
interface
   function checksum(block, nbytes) bind(c)
      use iso_c_binding
      type(*),intent(in)      :: block(*)
      integer(c_size_t),value :: nbytes
      integer(c_int64_t)      :: checksum
   end function checksum
end interface
⋮
integer(c_int64_t) blocksum
⋮
blocksum = checksum(x, c_sizeof(x))
```

Like `class(*)`, a `type(*)` variable is **unlimited polymorphic**, and has no declared type. But unlike `class(*)`, a `type(*)` dummy argument is not allowed to be associated with a type that has type parameters or a type-bound procedure part. Also, only dummy arguments can be declared `type(*)`.[1]

An assumed-type dummy argument is not allowed to be allocatable, a coarray, a pointer, or an explicit-shape array; the first three of these require structure beyond a simple address, and the last one is redundant with assumed size. Furthermore, it is not permitted to have intent `out` or the `value` attribute. However, it may be assumed shape or assumed rank, in which case it will interoperate with a C descriptor instead of a `void *` pointer; this is described in Section 21.6.2.

Because an assumed type dummy argument essentially has no type information associated with it, its usage within Fortran is extremely limited. It is only allowed to appear as an actual argument corresponding to a dummy argument that is also assumed type, or as the first argument of the intrinsic functions `present` and `rank`, and the argument of the function `c_loc` from the intrinsic module `iso_c_binding`. If it is an array, it is also permitted as the first argument of one of the intrinsic functions `is_contiguous`, `lbound`, `shape`, `size`, and `ubound`.

[1] This may also be done with an `implicit` statement.

An assumed-type variable that is not assumed shape or assumed rank must not be further passed on to an assumed-rank dummy argument. This is because in the scalar and assumed-size cases there is no type information available, but both assumed shape and assumed rank require type information.

The interface of a procedure with an assumed-type dummy argument is required to be explicit, even if it is not interoperable (that is, does not have the `bind(c)` attribute).

21.4 Assumed character length

One of the annoyances when calling C functions directly from Fortran is the need to remember to append a NUL character to the end when passing character strings, as the character length is not passed. Fortran 2018 addresses this by permitting an interoperable procedure to have a dummy argument with assumed character length.

Unfortunately, even in the simplest scalar case, this does not simply pass the character length as an extra argument, but causes a **C descriptor** to be passed by reference. This means that the C function needs to include the source file `iso_fortran_binding.h` to provide the definition of the C descriptor.

For example, a C function to write a Fortran character string to C's standard error file, with a newline character to end the line, could have the interface

```
interface
   subroutine err_msg(string) bind(c)
      use iso_c_binding
      character(*,c_char),intent(in) :: string
   end subroutine err_msg
end interface
```

and could be implemented by the C function

```
#include <stdio.h>
#include "iso_fortran_binding.h"
void err_msg(CFI_cdesc_t *string)
{
  int len;
  char *p = string->base_addr;
  for (len=string->elem_len; len>0; len--) putc(*p++,stderr);
  putc('\n',stderr);
}
```

When invoked by

```
call err_msg('oops')
```

the message '`oops`' would be written (with a newline).

C descriptors are far more powerful than this, and enable many other advanced techniques, but for simple string handling this is all we need. The next section describes C descriptors in full detail.

21.5 C descriptors

21.5.1 Introduction

In order to be able to handle Fortran objects that have no equivalent concept in C, Fortran 2018 defines **C descriptors**. This descriptor is a C structure containing all the information needed to describe the newly supported Fortran objects. The exact contents of the structure are processor dependent, though some members are fully standardized. The Fortran processor provides a source file `iso_fortran_binding.h` that contains the definition of this structure, together with macro definitions and function prototypes for a C function to use when interacting with the C descriptor.

The Fortran objects that C descriptors are used for are all dummy arguments,[2] and are assumed character length, assumed shape, assumed rank (Section 23.16), allocatable, or pointers. They must have interoperable type and kind type parameter (if any), except that an assumed-shape or assumed-rank dummy argument is also permitted to be assumed type (Section 21.6.2). An allocatable or pointer dummy argument must not be of a type with default initialization, and if of type character must have deferred character length.

The information about the object contained in the descriptor includes its address, rank, type, size (if scalar) or element size (if an array), and whether it is allocatable, a pointer, or neither. For an array the descriptor additionally contains information describing the shape, bounds (only relevant for allocatable and pointer), and memory layout; the latter is particularly important for arrays that are assumed shape, assumed rank, or pointers, as these can have discontiguous memory storage.

In order to preserve Fortran semantics, it is forbidden to access memory through the C descriptor that is not part of the object the descriptor describes; for example, memory that is before the object, after the object, or, when the object is discontiguous, in the gaps in the object.

Also, a C descriptor must not be modified, or even copied, except by using the functions provided by `iso_fortran_binding.h`; this is to preserve the integrity of the descriptor and any associated run-time data structures.

21.5.2 Standard members

A C descriptor is a C structure with type `CFI_cdesc_t`. Its first three members are:

`void * base_addr;` the address of the object; for allocatables and pointers, it is the address of the allocated memory or associated target. For an unallocated allocatable or a disassociated pointer, `base_addr` is a null pointer; otherwise, `base_addr` is not null, though for an object of zero size, it is not the address of any usable memory.

`size_t elem_len;` storage size in bytes of the object (if scalar) or an array element (if an array).

`int version;` version number of the descriptor. It is the value of `cfi_version` in the file `iso_fortran_binding.h` that defined the format and meaning of this C descriptor.

[2] Both for Fortran calling C (the dummy arguments in the interface), and for C calling Fortran.

A C descriptor also contains the following members, in any order, possibly interspersed with non-standard processor-dependent members.

`CFI_attribute_t attribute;` argument classification, indicating whether the object is allocatable, a pointer, or neither, according to Table 21.1.

`CFI_rank_t rank;` rank of the object (zero if scalar). The maximum rank supported is provided by the value of `CFI_MAX_RANK`.

`CFI_type_t type;` data type of the object; this is an integer type code from Table 21.2.

The last member in a C descriptor is

`CFI_dim_t dim[];` member describing the shape, bounds, and memory layout of an array object. This is described in Section 21.5.5.

The typedef names `CFI_attribute_t`, `CFI_rank_t`, and `CFI_type_t` are defined in `iso_fortran_binding.h`, and specify standard signed integer types, each with a size suitable for its purpose.

21.5.3 Argument classification (attribute codes)

The macros in Table 21.1 provide integer values for the `attribute` member of a C descriptor; they are non-negative and distinct.

The value passed in the `attribute` member when invoked from Fortran describes the dummy argument in the Fortran interface; thus, the only real use for this member is for error checking. For example, if the C function is expecting to be passed an allocatable array, it can check that the interface in the Fortran invocation really did specify that it was allocatable.

Table 21.1: Macros for attribute codes.

Macro name	Attribute
`CFI_attribute_allocatable`	allocatable
`CFI_attribute_pointer`	data pointer
`CFI_attribute_other`	non-allocatable non-pointer

21.5.4 Argument data type

The `type` member of a C descriptor specifies the data type of the actual argument in C terms. Unless the Fortran interface is assumed type, this will be the same as that of the dummy argument in the interface.

The first part of Table 21.2 lists the C type specifiers which have type codes of the form `CFI_type_`*type*, where *type* is the C type specifier (replacing internal spaces with underscore, and removing underscores at the beginnings of words); for example `CFI_type_int`,

Table 21.2: `CFI_type_` macros for type codes.

C type specifiers with `CFI_type_`*type* macros		
`signed char`	`intptr_t`	`int_least16_t`
`short`	`ptrdiff_t`	`int_least32_t`
`int`	`int8_t`	`int_least64_t`
`long`	`int16_t`	`int_fast8_t`
`long long`	`int32_t`	`int_fast16_t`
`size_t`	`int64_t`	`int_fast32_t`
`intmax_t`	`int_least8_t`	`int_fast64_t`
`float`	`double`	`long double`
`float _Complex`	`double _Complex`	`long double _Complex`
`_Bool`	`char`	

Additional `CFI_type_` macros

Macro name	Meaning
`CFI_type_cptr`	`void *` or any C data pointer
`CFI_type_struct`	interoperable C structure
`CFI_type_other`	Fortran non-interoperable type

`CFI_type_Bool`, and `CFI_type_long_double_Complex`. There are three additional type codes that do not fit this pattern, shown in the second part of the table.

All the `CFI_type_` type code macros have constant integer values. If a C type has no Fortran equivalent, its type code will be negative. A Fortran type with no C equivalent has the type code `CFI_type_other`, which is also negative and distinct from the other type codes. All other type codes are positive, but in general need not be distinct; for example, `CFI_type_long_long`, `CFI_type_int64_t`, `CFI_type_int_least64_t`, `CFI_type_int_fast64_t`, and `CFI_type_intmax_t` are likely to all be the same. The type code `CFI_type_struct` specifies an interoperable C structure (or Fortran `bind(c)` type); its value is distinct. Also, a Fortran intrinsic type that is interoperable with a C type that is not listed in the table, for example '`int128_t`', will have a positive value that is distinct.

21.5.5 Array layout information

Array information is provided by the `dim` member of a C descriptor; this is an array of structures with type `CFI_dim_t`, with one element for each rank of the array, the elements being in Fortran dimension order (first element is for the first Fortran dimension). For each dimension, the structure members are, in any order:

`CFI_index_t lower_bound;` the lower bound. This only has meaning for an array pointer or allocatable array; for all others, it has the value zero.

`CFI_index_t extent;` size of the dimension, or -1 for the final dimension of an assumed-size array.

`CFI_index_t sm;` spacing between elements, or 'memory stride'; that is, the difference in bytes between the addresses of successive elements in that dimension.

where `CFI_index_t` is a typedef name for a standard signed integer type capable of representing a memory address difference in bytes. The structure could have additional, non-standard, members.

21.6 Accessing Fortran objects

21.6.1 Traversing contiguous Fortran arrays

It is very straightforward to access every element of a **contiguous** Fortran array of any rank; all you need to do is to calculate the number of elements from the C descriptor, and then step through the elements. For example,

```
int64_t xor64(CFI_cdesc_t *cdesc) {
    int i;
    size_t nelts = 1;
    int64_t result = 0;
    int64_t *a = (int64_t *)cdesc->base_addr;
    for (i=0; i<cdesc->rank; i++) nelts *= cdesc->dim[i].extent;
    while (nelts-->0) result ^= *a++;
    return result;
}
```

computes the elementwise bitwise exclusive-or of a contiguous Fortran array with element storage size of 64 bits, first calculating the number of elements from the array extents, then initializing the result, and using bitwise exclusive-or assignment (`^=`) with every element. This C function could be used from Fortran on three-dimensional arrays by using the interface

```
interface
   pure function xor64(array) bind(c) result(r)
      use iso_c_binding
      integer(c_int64_t), intent(in), contiguous :: array(:,:,:)
      integer(c_int64_t) r
   end function xor64
end interface
```

Note that because the function assumes the array is contiguous, the `contiguous` attribute is necessary in the interface, so that a reference to the function from Fortran with a **discontiguous** actual argument will pass a contiguous copy of it to the function.

There is a utility function declared in `iso_fortran_binding.h` that can be used to check this assumption:

```
int CFI_is_contiguous(const CFI_cdesc_t *dv);
```

The argument `dv` must be the address of a descriptor that describes an array, and must not be an unallocated allocatable or disassociated pointer (so the `base_addr` member of the C descriptor must not be a null pointer). The result has the value 1 if the array described by `dv` is contiguous, and zero otherwise. In our previous example, we could use this to produce an error message if called with an inappropriate array:

```
if (!is_contiguous(cdesc)) {
    fputs("xor64 called with discontiguous array\n",stderr);
    exit(2);
}
```

Other than error checking, this function can be used to select a more efficient code path when a function that can handle discontiguous is called with a contiguous array argument; as we will see in Section 21.6.3, the code for traversing a discontiguous array is significantly more complicated with higher overheads.

Notice that the way the code inside the function is written, it would work without change on arrays of any rank, and even on scalars (though the result is not very interesting in the scalar case, being merely the argument value unchanged); see Section 21.6.6 for details of how this can be enabled.

21.6.2 Generic programming with assumed type

We have already seen, in Section 21.3, how **assumed-type** dummy arguments can be used for low-level C interoperability. When combined with assumed shape or assumed rank, however, assumed type works very differently; a C descriptor is passed, and this includes data type information that can be used by the invoked C procedure.

For example, the C function in Figure 21.1 could be invoked via the interface

```
interface
   subroutine negate_real(array) bind(c)
      type(*), intent(inout), contiguous :: array(:,:,:)
   end subroutine negate_real
end interface
```

to negate any three-dimensional real array.

As with the previous example, the `contiguous` attribute is used to avoid complicating the example with discontiguous array traversal code. Similarly, the code inside the function would work on scalars or arrays of any rank.

21.6.3 Traversing discontiguous Fortran arrays

Although the `contiguous` attribute is very useful for simplifying the objects that a C function needs to be able to handle, and the `CFI_is_contiguous` function can be used to check at run time that no mistake has been made, it is nearly always far more efficient to process a discontiguous array section directly than it is to make a copy of it (and potentially also copy it back afterwards). Fortunately, it is simple, though tedious, to traverse a discontiguous array using the information in the C descriptor.

Figure 21.1 A C function for negating a floating-point array.

```
void negate_real(CFI_cdesc_t *array) {
    size_t i;
    size_t nelts = 1;
    for (i=0; i<cdesc->rank; i++) nelts *= cdesc->dim[i].extent;
    if (cdesc->type==CFI_type_float) {
        float *f = cdesc->base_addr;
        for (i=0; i<nelts; i++) f[i] = -f[i];
    }
    else if (cdesc->type==CFI_type_double) {
        double *d = cdesc->base_addr;
        for (i=0; i<nelts; i++) d[i] = -d[i];
    }
    else if (cdesc->type==CFI_type_long_double) {
        long double *ld = cdesc->base_addr;
        for (i=0; i<nelts; i++) ld[i] = -ld[i];
    }
    else {
        fputs("negate_real: not a supported real type\n",stderr);
        exit(2);
    }
}
```

The simplest way to do this is to have one pointer per array dimension. For example, with a three-dimensional array we could have a 'plane pointer' starting at the base address, being stepped by the memory stride of a whole plane, and inside that loop a 'column pointer' starting at the beginning of the current plane, being stepped by the memory stride of a whole column, and inside that loop an 'item pointer' starting at the beginning of the current column and being stepped by the memory stride of the innermost dimension.

This does have a lot of overhead, though, and accessing the extents and memory strides through a pointer is not conducive to optimization (for example, keeping these values and pointers in registers).

A better way is to use a single pointer to traverse the whole array, with precomputed step values, and copying the extents into local variables.

Figure 21.2 shows this method, extending the `int64_t` exclusive-or reduction we saw in Section 21.6.1 to handle discontiguous three-dimensional arrays. The interface for invoking this function from Fortran is the same as before, except that the `contiguous` attribute is omitted from the declaration of the dummy argument `array`.

The precomputed step values are derived straightforwardly from the memory strides (`sm` values) for each dimension. For the first (innermost) dimension, the step value $\texttt{step}_1$ is just the same as the memory stride $\texttt{sm}_1$, while for every subsequent dimension $i+1$, the step value is the memory stride of that dimension minus the accumulated increments of all the inner dimensions, which is $\texttt{sm}_i \times \texttt{extent}_i$.

Figure 21.2 Assumed-shape array traversal.

```
int64_t xor64(CFI_cdesc_t *cdesc) {
    CFI_extent i,j,k,step0,step1,step2,extent0,extent1,extent2;
    int64_t result = 0;
    char *ptr = cdesc->base_addr;
    step0 = cdesc->dim[0].sm;
    extent0 = cdesc->dim[0].extent;
    step1 = cdesc->dim[1].sm - cdesc->dim[0]*extent0;
    extent1 = cdesc->dim[1].extent;
    step2 = cdesc->dim[2].sm - cdesc->dim[1].sm*extent1;
    extent2 = cdesc->dim[2].extent;
    for (k=0; k<extent2; k++) {
        for (j=0; j<extent1; j++) {
            for (i=0; i<extent0; i++) {
                result ^= *(int64_t *)ptr;
                ptr += step0;
            }
            ptr += step1;
        }
        ptr += step2;
    }
    return result;
}
```

Optimization should keep the pointer and the `step` and `extent` values in registers, so despite the increased overhead compared with a contiguous array traversal, with a simple operation like exclusive-or the performance is likely to be dominated by the time taken to access the array from main memory.

We can also see that both the simplistic and this faster formulation of discontiguous traversal will only work for three-dimensional arrays, whereas the contiguous case worked for any rank of array.

21.6.4 Fortran pointer operations

Accessing a Fortran pointer is relatively easy, using the `base_addr` member of the C descriptor in conjunction with the array information if it is an array. For example, testing whether a Fortran pointer argument is disassociated is simple:

```
if (cdesc->base_addr)
    printf("Associated\n");
else
    printf("Disassociated\n");
```

For nullifying or pointer assigning, a function is provided by `iso_fortran_binding.h`:

```
int CFI_setpointer(CFI_cdesc_t *result, CFI_cdesc_t *source,
                   const CFI_index_t lower_bounds[]);
```

where `result` must be a C descriptor for a Fortran pointer, and `source` must be a null pointer or a C descriptor for either a similar pointer to `result` or an acceptable target for `result`. That is, if `source` is not a null pointer, its `type`, `elem_len`, and `rank` members must be the same as that of `result`. If `source` does not describe an array, `lower_bounds` is ignored; otherwise, it may be a null pointer or the address of an array of size `source->rank`.

The result of the function is an **error indicator**; this is similar to a `stat=` value, being zero if the function executed successfully, and nonzero otherwise (Section 21.6.8).

If `source` is a null pointer, the effect is the same as the Fortran statement `nullify(result)`, and `result` becomes a disassociated pointer.

If `source` is not a null pointer, the effect is similar to the Fortran pointer assignment `result => source`. If `source` is a C descriptor for a Fortran pointer, `result` will have the same pointer association as `source` (either disassociated, or associated with the same target). Otherwise, `result` will become associated with the object described by `source`. Note that if this object is a Fortran entity, it must have the `target` attribute.

If `source` describes an array and `lower_bounds` is not a null pointer, the lower bounds of `result` are set to the values specified in `lower_bounds`, similar to the Fortran pointer assignment `result(`$\texttt{lower}_1$`:,`$\texttt{lower}_2$`:,...) => source` (as described in Section 7.15).

This simple example chooses between two scalar integer pointers, choosing the non-null pointer that references the highest value; if an error occurs the result will be nullified.

```
interface
   subroutine set_ptr_to_max(result, inptr1, inptr2) bind(c)
      use iso_c_binding
      integer(c_int), pointer, intent(out) :: result
      integer(c_int), pointer, intent(in) :: inptr1, inptr2
   end subroutine set_ptr_to_max
end interface

#include "iso_fortran_binding.h"
void set_ptr_to_max(CFI_cdesc_t *result,CFI_cdesc_t *inptr1,
                    CFI_cdesc_t *inptr2) {
    int res;
    if (!inptr1->base_addr)
        res = CFI_setpointer(result,inptr2,(CFI_index_t*)0);
    else if (!inptr2->base_addr ||
             *(int *)inptr1->base_addr>=*(int *)inptr2->base_addr)
        res = CFI_setpointer(result,inptr1,(CFI_index_t*)0);
    else
        res = CFI_setpointer(result,inptr2,(CFI_index_t*)0);
    if (res)
        (void)CFI_setpointer(result,(CFI_cdesc_t*)0,(CFI_index_t*)0);
}
```

Functions are also available for allocating and deallocating Fortran pointers via their C descriptors. These functions also operate on allocatable variables, so are described in the next section.

21.6.5 Allocatable objects

Like Fortran pointers, accessing Fortran allocatable objects using the `base_addr` member of the C descriptor is straightforward. For example, testing whether a Fortran allocatable argument is allocated is simple:

```
if (cdesc->base_addr)
    printf("Allocated\n");
else
    printf("Unallocated\n");
```

Deallocation is also very simple, using the function

```
int CFI_deallocate(CFI_cdesc_t *dv);
```

where `dv` is a C descriptor for a Fortran allocatable or pointer argument. The return value is an error indicator.

The memory must have been allocated using the Fortran memory allocator, that is, by the function `CFI_allocate` or a Fortran `allocate` statement. If deallocation is successful, the `base_addr` member of `dv` will become a null pointer.

Note that in the pointer case the conditions for successful deallocation by a Fortran `deallocate` statement must be satisfied. For example, the pointer must be associated with the entirety of an allocated object, with the array elements in the same order.

For allocation, array bounds and character length may have to be supplied. This is done by the function `CFI_allocate`:

```
int CFI_allocate(CFI_cdesc_t *dv, const CFI_index_t lower_bounds[],
                 const CFI_index_t upper_bounds[], size_t elem_len);
```

where `dv` is a C descriptor for an unallocated allocatable or disassociated pointer. If the object is an array, `lower_bounds` and `upper_bounds` must be arrays of size `dv->rank`; these arguments are ignored for a scalar. If the object is a Fortran character type, `elem_len` is the size of each element in bytes, and this overrides `dv->elem_len`; otherwise, `elem_len` is ignored. The return value is an error indicator.

For example, the C function in Figure 21.3, if invoked with the interface

```
interface
   subroutine get_home(home) bind(c)
      use iso_c_binding
      character(:,c_char), allocatable, intent(inout) :: home
   end subroutine get_home
end interface
```

will allocate a deferred-length character scalar to hold the value of the environment variable HOME (usually the user's home directory path), deallocating it first if necessary. On any error, or if the environment variable does not exist, the argument will become unallocated.

Figure 21.3 A C function that allocates a Fortran character variable.

```
#include <stdlib.h>
void get_home(CFI_cdesc_t *home) {
    char *homename = getenv("HOME");
    if (home->base_addr) {
        r = CFI_deallocate(home);
        if (r) return; /* failed. */
    }
    if (homename) {
        int r;
        size_t len = strlen(homename);
        r = CFI_allocate(home,(CFI_index_t*)0,(CFI_index_t*)0,len);
        if (r) return; /* failed. */
        memcpy(r->base_addr,homename,len);
    }
}
```

Finally, if the interface for a C function has an allocatable dummy argument with intent `out`, calling that function from Fortran with an allocated actual argument will automatically deallocate the argument before invocation. Similarly, if a Fortran procedure with an intent `out` allocatable dummy argument is called with an allocated actual argument from C, the argument will be automatically be deallocated on entry to the procedure. This preserves Fortran semantics for allocatable intent `out` dummy arguments whenever either procedure in a call is Fortran.

21.6.6 Handling arrays of any rank

We saw in Section 21.6.1 that the C code for elementwise processing of a contiguous array can be the same for any rank, but that the Fortran interface is different. The new **assumed rank** feature allows us to write a single interface for a C function that accepts any rank of array. This feature is described fully in Section 23.16, but all we need here is the declaration for the interface.

For example, an interface that would let us call the earlier `xor64` example with an array of any rank is

```
interface
   pure function xor64(array) bind(c) result(r)
      use iso_c_binding
      integer(c_int64_t), intent(in), contiguous :: array(..)
      integer(c_int64_t) r
   end function xor64
end interface
```

The only change to the interface is the use of '`..`' to denote assumed rank.

This may be combined with the assumed type feature for generic programming, in the same way. So the interface

```
interface
   subroutine negate_real(array) bind(c)
      type(*), intent(inout), contiguous :: array(..)
   end subroutine negate_real
end interface
```

would let us negate a real array of any rank.

There is one fly in this ointment: if the dummy argument is assumed rank and not assumed type, the actual argument is allowed to be an assumed-size array. This is a problem because an assumed-size array has no upper bound in the last dimension, rendering the calculation of the number of elements impossible. To avoid incorrect results or attempting to process all memory (and likely crashing), this should be checked for and an appropriate action taken.

This problem does not arise for `negate_real`, because its array argument is assumed type, but is a problem for `xor64`. To detect an assumed-size argument at run time, it suffices to check for the final extent of the C descriptor being less than zero (and this should be done before attempting to calculate the number of elements). For example,

```
if (cdesc->rank>0 && cdesc->dim[cdesc->rank-1].extent<0) {
    fputs("Actual argument to xor64 is assumed-size\n",stderr);
    exit(2);
}
```

When it comes to discontiguous arrays of any rank, the interface remains simple – just leave off the `contiguous` attribute – but the traversal is more complicated. If the intention is only to handle, say, arrays up to rank five, a reasonable solution is to use the method shown in Figure 21.2 with a deep loop nest, pretending that any missing dimension has extent one.

A better way is to use a single loop together with an array of indexes to keep track of how much of each dimension has been traversed. Figure 21.4 shows how this can be done. The dimension-collapsing phases are not actually necessary, but improve performance by reducing the bookkeeping needed during traversal. This is especially important for the innermost dimensions, since that is the only loop that could be vectorized. If the array is contiguous, dimension collapse will reduce the internal array description to rank one, and only the innermost loop will execute more than once.

21.6.7 Accessing individual array elements via a C descriptor

It is not difficult to manually calculate the address of a single element of an array that is passed as a C descriptor, using subscripts; simply subtract the lower bounds from the subscripts, multiply each by its 'memory stride' (`sm` member), and add to the base address cast to a `char *` pointer. For a non-allocatable non-pointer dummy argument, the lower bounds in a C function are always zero, so the subtraction can be omitted in that case.

For example, to access an element `a(i, j)` for a non-allocatable non-pointer dummy argument `a` passed as the C descriptor `a`, the address is simply

```
(i*a->dim[0].sm + j*a->dim[1].sm) + (char *)(a->base_addr)
```

Figure 21.4 Traversing a discontiguous array of any supported rank.

```
int64_t xor64(CFI_cdesc_t *cdesc) {
    int i;
    CFI_extent extent[CFI_MAX_RANK],ii[CFI_MAX_RANK],ix,step[CFI_MAX_RANK];
    int64_t result = 0;
    int rank = cdesc->rank;
    char *ptr = cdesc->base_addr;
    if (rank==0) return *(int64_t *)ptr; /* Scalar result is trivial. */
    if (cdesc->dim[rank-1]<0) {
        fputs("Actual argument to xor64 is assumed-size\n",stderr);
        exit(2);
    }
    /* Establish step values for the array, and save the extents locally. */
    step[0] = cdesc->dim[0].sm;
    extent[0] = cdesc->dim[0].extent;
    for (i=1; i<rank; i++) {
        ii[i] = 0;
        step[i] = cdesc->dim[i].sm - cdesc->dim[i-1].sm*extent[i-1];
        extent[i] = cdesc->dim[i].extent;
        if (extent[i]==0) return 0; /* Zero-sized array has zero result. */
    }
    /* Collapse outermost and innermost contiguous dimensions. */
    while (step[rank-1]==0) {
        extent[rank-2] *= extent[rank-1];
        rank--;
    }
    while (rank>2 && step[1]==0) {
        extent[0] *= extent[1];
        rank--;
        for (i=1; i<rank; i++) {
            extent[i] = extent[i+1];
            step[i] = step[i+1];
        }
    }
    /* Now traverse the array. */
    for (;;) {
        for (ix=0; ix<extent[0]; ix++,p+=step[0]) {
            result ^= *(int64_t *)ptr;
        }
        /* Step through the outer dimensions. */
        for (i=1; i<rank; i++) {
            ii[i]++;
            if (ii[i]<extent[i]) break;
            ii[i] = 0;
        }
        if (i==rank) return result;
    }
}
```

For accessing an element of an array that is allocatable or a pointer, with subscripts based on the actual lower bounds, the complete formulation

```
((i-a->dim[0].lower)*a->dim[0].sm +
 (j-a->dim[1].lower)*a->dim[1].sm) + (char *)(a->base_addr)
```

will be necessary.

A convenience function is available in `iso_fortran_binding.h` to simplify this:

```
void * CFI_address(const CFI_cdesc_t *dv,const CFI_index_t subs[]);
```

where `dv` is a C descriptor for an object other than an unallocated allocatable or disassociated pointer, and `subs` is an array of subscripts of size `dv->rank`; the address of the array element is returned. If `dv` is scalar, `subs` is ignored and may be a null pointer. As in the manual case, the subscripts must be within the bounds of the array and, if the array is assumed size, must specify an element within the size of the array. However, as there is no error handling provided by `CFI_address`, it has little advantage over the manual method, unless the rank is high or the subscripts are already in an array of type `CFI_index_t`.

Once the address has been calculated, whether manually or using `CFI_address`, referencing or defining the element will also need to cast the address to the data type of the array. For example, the C function

```
void zero_element(CFI_cdesc_t *a,int i,int j) {
    char *elt = a->base_addr;
    elt += (i*a->dim[0].sm + j*a->dim[1].sm);
    switch (a->type) {
    case CFI_float:
        *(float *)elt = 0.0f;
        break;
    case CFI_double:
        *(double *)elt = 0.0;
        break;
    case CFI_long_double:
        *(long double *)elt = 0.0L;
        break;
    default:
        fputs("zero_element: unsupported data type\n",stderr);
        exit(2);
    }
}
```

could be invoked through the interface

```
interface
   subroutine zero_element(a,i,j) bind(c)
      use iso_c_binding
      type(*),intent(inout) :: a(:,:)
      integer(c_int),value :: i,j
   end subroutine zero_element
end interface
```

to set a selected element of a floating-point array to zero.

21.6.8 Handling errors from CFI functions

Many of the functions provided by `iso_fortran_binding.h` return success or failure as an **error indicator**. This is an integer value akin to the `stat=` specifier of a Fortran statement or `stat` argument of some intrinsic procedures, and is zero to indicate that the function was successful (no error), and nonzero to indicate which error was detected. If an error was detected, the function will not have modified any of its arguments.

The macro `CFI_SUCCESS` is defined to be zero for use in testing for success. Standard error conditions, with corresponding macro names, are shown in Table 21.3. All values are nonzero and distinct. If a processor can detect additional error conditions, they will return values that differ from the standard ones.

Table 21.3: Macros for error codes

Macro name	Error condition
`CFI_ERROR_BASE_ADDR_NULL`	The `base_addr` member of a C descriptor is a null pointer, but a non-null pointer is required.
`CFI_ERROR_BASE_ADDR_NOT_NULL`	The `base_addr` member of a C descriptor is not a null pointer, but a null pointer is required.
`CFI_INVALID_ELEM_LEN`	Invalid value for `elem_len` member of a C descriptor.
`CFI_INVALID_RANK`	Invalid value for `rank` member of a C descriptor.
`CFI_INVALID_TYPE`	Invalid value for `type` member of a C descriptor.
`CFI_INVALID_ATTRIBUTE`	Invalid value for `attribute` member of a C descriptor.
`CFI_INVALID_EXTENT`	Invalid value for `extent` member of a `CFI_dim_t` structure.
`CFI_INVALID_DESCRIPTOR`	A C descriptor is invalid in some way.
`CFI_ERROR_MEM_ALLOCATION`	Memory allocation failed.
`CFI_ERROR_OUT_OF_BOUNDS`	A reference is out of bounds.

21.7 Calling Fortran with C descriptors

21.7.1 Allocating storage for a C descriptor

When a C function is called from Fortran, the Fortran processor takes care of setting up the C descriptor. But when calling Fortran from a C function, unless simply passing on an argument received from Fortran, storage will need to be allocated for the descriptor itself.

To allocate the storage for a C descriptor, the macro `CFI_CDESC_T` is provided. It is a function-like macro that takes one argument and evaluates to an unqualified type of suitable size and alignment for holding a C descriptor. The argument must be an integer constant expression that specifies the rank of the object to be described. The requirement for the rank to be constant means that if you want to be able to handle an array of any rank, the maximum possible rank would need to be specified here, for example

```
CFI_CDESC_T(CFI_MAX_RANK) pointer_to_any_array;
```

To use the allocated storage, it is usually necessary to cast the address of the storage to the descriptor type `CFI_cdesc_t *`, for example

```
CFI_CDESC_T(5) object;
int stat;
CFI_cdesc_t * dv = (CFI_cdesc_t *)&object;
stat = CFI_establish(dv, ...); /* See Section 21.7.2. */
```

The C descriptor may be allocated statically, on the stack, or it may be a component of a structure which might then be allocated on the heap, for example:

```
struct fortran_string_list_item {
    CFI_CDESC_T(0) string
    struct fortran_string_list_item *next;
};
```

Before any C descriptor is used, whether its memory was allocated statically or on the stack by `CFI_CDESC_T`, or on the heap as part of a structure allocated by `malloc`, it must be established.

21.7.2 Establishing a C descriptor

Before a C descriptor can be used, it must be **established**; this sets up its internal data, both standard members and any processor-dependent members. When a C function is invoked from Fortran, the Fortran processor establishes the C descriptors for the relevant arguments, but when a Fortran processor is invoked from C, the C function needs to do this. This is done by the function `CFI_establish`, which has the prototype

```
int CFI_establish(CFI_cdesc_t *dv, void *base_addr,
                  CFI_attribute_t attribute, CFI_type_t type,
                  size_t elem_len, CFI_rank_t rank,
                  const CFI_index_t extents[]);
```

where `dv` is the C descriptor to be established; it must not be associated with a Fortran dummy or actual argument, and it is highly recommended that the `CFI_CDESC_T` macro be used to allocate its storage. The remaining arguments provide the characteristics of the object that the C descriptor will describe, and the return value is an error indicator.

`base_addr` the address of the object, or a null pointer.

`attribute` whether the object is allocatable, a pointer, or an ordinary data object.

`type` a type code in Table 21.2 or the type code of an interoperable C type.

`elem_len` provides the size in bytes of the object, or an element of the object when it is an array, if `type` identifies a structure or character type, and is ignored otherwise.

`rank` specifies the desired rank.

`extents` is an array of size `rank` of non-negative values specifying the array shape, when `rank` is non-zero and `base_addr` is not a null pointer; otherwise, it is ignored.

When `attribute` is `CFI_attribute_allocatable`, `base_addr` must be a null pointer signifying an unallocated allocatable. When `attribute` is `CFI_attribute_pointer`, `base_addr` may be a null pointer to signify a disassociated Fortran pointer, or the address of a suitable object for the C descriptor to be pointer-associated with. When `attribute` is `CFI_attribute_other`, `base_addr` may be a null pointer signifying that the descriptor is unfinished (it must be completed by `CFI_section` or `CFI_select_part` before use), or the address of the object described by the descriptor.

When the object is an array and `base_addr` is not a null pointer, the lower bounds are zero.

This looks a bit complicated, but it is not too hard to use it to construct C descriptors for calling Fortran procedures that have assumed-shape or assumed character length arguments. For example, given the C array and string

```
float x[10][20];
char *name;
```

and the Fortran procedure with the interface

```
interface
   subroutine process(name,a) bind(c)
      use iso_c_binding
      character(*,c_char),intent(in) :: name
      real(c_float) a(:,:)
   end subroutine
end subroutine
```

we can pass `x` and `name` to the assumed-shape array and assumed-length character arguments of the Fortran procedure with

```
extern void process(CFI_cdesc_t *,CFI_cdesc_t *);
CFI_CDESC_T(2) xdesc;
CFI_CDESC_T(0) namedesc;
CFI_index_t xshape[2] = { 20,10 };
if (CFI_establish((CFI_cdesc_t *)&xdesc,&x,CFI_attribute_other,
                  CFI_type_float,0,2,xshape) ||
    CFI_establish((CFI_cdesc_t *)&namedesc,name,CFI_attribute_other,
                  CFI_type_char,strlen(name),0,0)) {
    fputs("Unexpected CFI_establish failure\n",stderr);
    exit(2);
}
process((CFI_cdesc_t *)&namedesc,(CFI_cdesc_t *)&xdesc);
```

21.7.3 Constructing an array section

In Fortran a section of an array, possibly discontiguous, can be passed as an actual argument using array section notation. This facility is also available from C when the called procedure expects a C descriptor, using the `CFI_section` function. This function needs to have an established C descriptor that will be modified to describe the section, a C descriptor that describes the base array, and all the information to describe the array section.

```
int CFI_section(CFI_cdesc_t *result, const CFI_cdesc_t *source,
                const CFI_index_t lower[], const CFI_index_t upper[],
                const CFI_index_t stride[]);
```

where `result` is an established C descriptor that will be modified to describe the section, `source` is a C descriptor that describes the base array, and `lower`, `upper`, and `stride` are arrays of size `source->rank` that describe the section. The return value is an error indicator.

The `lower`, `upper`, and `stride` arrays may be null pointers to take the default for all dimensions: the lower bounds and upper bounds from `source`, and the value 1 for the stride. Note that `upper` must not be null when the base array is associated with an assumed-size array. The `type` and `elem_size` members of `result` must be the same as those of `source`.

Note that `result` is not permitted to be allocatable (that is, its `attribute` member must not be equal to `CFI_attribute_allocatable`), because allocatable arrays are always contiguous and array sections are usually discontiguous. If `result` is a pointer, the effect is that it is pointer-associated with the section, otherwise the effect is as if it were argument-associated with the section. Note that `result` must not be currently associated with a Fortran actual or dummy argument unless it is a pointerin which case `CFI_section` acts like pointer assignment.[3]

The section described can be characterized in Fortran as

`source(`$\texttt{lower}_1$`:`$\texttt{upper}_1$`:`$\texttt{stride}_1$`, ...)`

where if $\texttt{stride}_i$ is zero, $\texttt{lower}_i$ must be equal to $\texttt{upper}_i$ and that dimension acts as if specified with the simple subscript $\texttt{lower}_i$. The rank of this array section is the number of dimensions with nonzero `stride`, and `result` must have this rank.

A simple example without error checking is:

```
extern void process_vector(CFI_cdesc_t *);
static int64_t x[200][100];
⋮
CFI_CDESC_T(2) xdesc;
CFI_CDESC_T(1) rowdesc;
CFI_index_t lower[2],upper[2],stride[2],xshape[2];
xshape[0] = 100;
xshape[1] = 200;
(void)CFI_establish((CFI_cdesc_t *)&xdesc,&x,CFI_attribute_other,
                    CFI_type_int64_t,0,2,xshape);
(void)CFI_establish((CFI_cdesc_t *)&rowdesc,0,CFI_attribute_other,
                    CFI_type_int64_t,0,1,0);
lower[0] = upper[0] = 99; stride[0] = 0;
lower[1] = 0; upper[1] = 22; stride[1] = 1;
(void)CFI_section((CFI_cdesc_t *)&rowdesc,(CFI_cdesc_t *)&xdesc,
                  lower,upper,stride);
process_vector((CFI_cdesc_t *)&rowdesc);
```

This example passes the first 23 elements of the 100th Fortran row (C column) of `x` to `process_vector` via a C descriptor for a discontiguous vector. It is notable that even this simple example has a lot of tedious bookkeeping that is easy to get wrong.

[3] Using this function to do pointer assignment instead of `CFI_setpointer` is opaque and not recommended.

The `process_vector` procedure would have a Fortran interface like

```
interface
   subroutine process_vector(array) bind(c)
      use iso_c_binding
      integer(c_int64_t), intent(inout) :: array(:)
   end subroutine
end interface
```

A more interesting example is shown in Figure 21.5. This is a C function that accepts an assumed-shape 64-bit integer array of rank three, and invokes the same Fortran interoperable procedure `process_vector` on every row of the third plane of that array.

Figure 21.5 Constructing and passing array sections with C descriptors.

```
extern void process_vector(CFI_cdesc_t *);
void process_rows_of_plane_3(CFI_cdesc_t *src) {
    CFI_index_t i,low[3],upper[3],stride[3],rowshape[1];
    CFI_CDESC_T(1) row;
    if (src->attribute!=CFI_attribute_other || src->rank!=3 ||
        src->dim[2].upper<3 || src->type!=CFI_type_int64_t || ) {
        fputs("Wrong class/rank/arrayclass/type of src\n",stderr);
        exit(2);
    }
    rowshape[0] = src->dim[1].extent; /* Size of row is no of cols */
    if (CFI_establish((CFI_cdesc_t *)&row,0,CFI_attribute_other,
                      CFI_type_int64_t,0,1,rowshape) {
        fputs("Unexpected CFI_establish error\n",stderr);
        exit(2);
    }
    stride[0] = 0;
    low[1] = src->dim[1].lower_bound;
    upper[1] = src->dim[1].upper_bound;
    stride[1] = 1;
    low[2] = upper[2] = 2; stride[2] = 0;
    for (i=0; i<src->dim[0].extent; i++) {
        low[0] = upper[0] = i;
        if (CFI_section((CFI_cdesc_t *)&row,src,low,upper,stride)) {
            fputs("Unexpected CFI_section error\n",stderr);
            exit(2);
        }
        process_vector((CDI_cdesc_t *)&row);
    }
}
```

21.7.4 Accessing components

In Fortran, an array section can also be constructed by taking a component, substring, or complex part of a parent array, for example

```
array%re
```

This is also possible from C when producing a C descriptor with the `CFI_select_part` function, which has the prototype:

```
int CFI_select_part(CFI_cdesc_t *result, const CFI_cdesc_t *source,
                    size_t offset, size_t elem_len);
```

It returns an error indicator, and its arguments are as follows:

`result` is an established C descriptor with the same rank as `source`, and will be updated to describe the array section. The type information is unchanged, so this must already describe the type of the component or part being selected. As with `CFI_section`, it must not be allocatable, and if it is currently associated with a Fortran argument, must be a pointer (and as with `CFI_section`, this is not recommended).

`source` is the C descriptor that describes the base array; it must not be unallocated, disassociated, or assumed size.

`offset` is the distance in bytes between the beginning of each array element and the component or part being described. If the base array is of an interoperable C structure, the `offsetof` macro may be used to obtain the correct value.

`elem_len` is the storage size in bytes of the part being selected if it is a Fortran character type, and this will override the information in `source`; otherwise, `elem_len` is ignored.

The part being selected must be a component, substring, or the real or imaginary part of a complex variable, and all sizes and the offset must be valid; for example, the selected part must lie entirely within each array element and must be properly aligned for its type.

This is most useful for interoperable structures; for example,

```
typedef struct {
    double r,i,j,k;
} Quaternion;
void process_j(CFI_cdesc_t *q) {
    CFI_CDESC_T(CFI_MAX_RANK) qj;
    CFI_index_t extent e[CFI_MAX_RANK];
    int i;
    for (i=0; i<q->rank; i++) e[i] = q->dim[i].extent;
    (void)CFI_establish((CFI_cdesc_t *)&qj,0,CFI_attribute_other,
                        CFI_type_double,0,q->rank,e);
    (void)CFI_select_part((CFI_cdesc_t *)&qj,q,
                          offsetof(Quaternion,j),0);
    process_doubles((CFI_cdesc_t *)&qj);
}
```

constructs a C descriptor (without error checking) for the array section that is the `j` member of a `Quaternion` array, and passes it to another function for processing.

21.8 Restrictions

21.8.1 Other limitations on C descriptors

If the address of a C descriptor is a formal parameter that corresponds to a Fortran actual argument or a C actual argument that corresponds to a Fortran dummy argument,

- the C descriptor must not be modified if either the corresponding dummy argument in the Fortran interface has intent `in` or the C descriptor is for a non-allocatable non-pointer object, and
- the `base_addr` member of the C descriptor must not be accessed before it is given a value if the corresponding dummy argument in the Fortran interface is a pointer and has intent `out`.

If a C descriptor is passed to an assumed-shape Fortran dummy argument, the C descriptor must have the correct rank and must not describe an assumed-size array.

21.8.2 Lifetimes of C descriptors

A C descriptor or C pointer that is associated with any part of a Fortran object becomes undefined in the same circumstances that would cause a Fortran pointer to it to become undefined (for example, if it is deallocated, or if it is an unsaved local variable and its containing procedure returns).

A C descriptor whose address is a formal parameter that corresponds to a Fortran dummy argument becomes undefined on return from a call to the function from Fortran. Furthermore, if the dummy argument does not have either the `target` or `asynchronous` attribute, all C pointers to any part of the object become undefined on return from the call.

No further use may be made of a C descriptor or pointer that becomes undefined in any of these ways.

21.9 Miscellaneous C interoperability changes

21.9.1 Interoperability with the C type ptrdiff_t

The named integer constant `c_ptrdiff_t` has been added to the `iso_c_binding` module for use as a kind parameter for type integer to allow interoperability with the C type `ptrdiff_t`.

21.9.2 Relaxation of interoperability requirements

Several of the restrictions for procedures in the `iso_c_binding` module that limited operation to interoperable entities have been lifted.

- The argument to `c_loc` may be a non-interoperable array. Being non-interoperable, no use of the data can be made from C, but a Fortran pointer to it can be recovered

by `c_f_pointer`, which now allows the `fptr` argument to be an array pointer of non-interoperable type. (The purpose here is to facilitate exchange of information between two Fortran procedures when the intervening procedure is C.)

- The argument to `c_funloc` may be a non-interoperable procedure, and the `fptr` argument to `c_f_procpointer` may be a non-interoperable procedure pointer. (As in the previous item, the purpose here is to facilitate information passing between two Fortran procedures via C.)

Also, the `cptr` argument to `c_f_pointer` may be the C address of a storage sequence that is not in use by any other Fortran entity. In this case, the `fptr` argument becomes associated with that storage sequence. (The purpose of this is to allow for interaction with memory allocators written in C.)

22. Fortran 2018 conformance with ISO/IEC/IEEE 60559:2011

22.1 Introduction

A large number of changes to the intrinsic modules `ieee_arithmetic`, `ieee_exceptions`, and `ieee_features` have been made for conformance with the new IEEE standard for floating-point arithmetic (ISO/IEC/IEEE 60559:2011).

22.2 Subnormal values

The new IEEE standard uses the term 'subnormal' instead of 'denormal' for a value with magnitude less than any normal value and less precision.

The named constants `ieee_negative_subnormal` and `ieee_positive_subnormal` of type `ieee_class_type` have been added to the module `ieee_arithmetic` with the same values as `ieee_negative_denormal` and `ieee_positive_denormal`, respectively.

The named constant `ieee_subnormal` of type `ieee_features_type` has been added to the module `ieee_features` and has the same value as `ieee_denormal`.

The inquiry function `ieee_support_subnormal` has been added to `ieee_arithmetic` and is exactly the same as `ieee_support_denormal` except for its name.

22.3 Type for floating-point modes

The type `ieee_modes_type` has been added to `ieee_exceptions` for storing all the floating-point modes, that is, the rounding modes, underflow mode, and halting mode. Also added to `ieee_exceptions` are subroutines for getting and setting the modes.

`call ieee_get_modes(modes)` where `modes` is a scalar of type `ieee_modes_type` that has intent `out` and is assigned the value of the floating-point modes.

`call ieee_set_modes(modes)` where `modes` is a scalar of type `ieee_modes_type` that has intent `in` and has a value obtained by a previous call of `ieee_get_modes`. The floating-point modes are restored to their values at the time of the previous call.

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

22.4 Rounding modes

The new IEEE standard has two modes for rounding to the nearest representable number, which differ in the way they handle ties: in the case of a tie, roundTiesToEven uses the value with an even least significant digit and roundTiesToAway uses the value further from zero. The value `ieee_nearest` of the type `ieee_round_type` corresponds to roundTiesToEven and the value `ieee_away` has been added to correspond to roundTiesToAway.

The subroutines `ieee_get_rounding_mode` and `ieee_set_rounding_mode` have an additional optional argument `radix` to allow the decimal rounding mode to be inquired about and set independently of the binary rounding mode; the new descriptions are as follows.

`call ieee_get_rounding_mode (round_value[, radix])` where:

`round_value` is scalar, of type `type(ieee_round_type)`, with intent `out`. It is assigned the floating-point rounding mode for the specified radix; the value will be `ieee_nearest`, `ieee_to_zero`, `ieee_up`, `ieee_down`, or `ieee_away` if one of the IEEE modes is in operation, and `ieee_other` otherwise.

`radix` is a scalar integer with intent `in`. It must have the value 2 or 10; if absent, the specified radix is 2.

`call ieee_set_rounding_mode (round_value[, radix])` where:

`round_value` is scalar, of type `type(ieee_round_type)`, with intent `in`. It specifies the mode to be set for the specified radix.

`radix` is a scalar integer with intent `in`. It must have the value 2 or 10; if absent, the specified radix is 2.

The subroutine must not be called unless the value of `ieee_support_rounding (round_value, x)` is true for some `x` with the specified radix such that the value of `ieee_support_datatype(x)` is true.

22.5 Rounded conversions

The elemental function `ieee_real` has been added to `ieee_arithmetic` for rounded conversion to real type, as specified in the IEEE standard by the convertFromInt and convertFormat operations.

`ieee_real (a[, kind])` converts the value of `a`, which must be of type integer or real, to type real rounded according to the current rounding mode. If `kind` is absent, the result is default real; otherwise the result is of type real with kind type parameter `kind`, which must be a scalar integer constant expression. The kind of the result, and of `a` if `a` is of type real, must be an IEEE real kind.

The elemental function `ieee_rint` has an extra optional argument:

`ieee_rint (x[, round])` where `round` is of type `ieee_round_type`, returns the value of `x` rounded to an integer according to the mode specified by `round`; if `round` is absent, the current rounding mode is used.

22.6 Fused multiply-add

The elemental function `ieee_fma` has been added to `ieee_arithmetic` for the fused multiply-add operation.

`ieee_fma (a, b, c)` where `a`, `b`, and `c` are real with the same kind type parameter returns the mathematical value of `a`×`b`+`c` rounded according to the rounding mode; `ieee_overflow`, `ieee_underflow`, and `ieee_inexact` are signaled according to the final step in the calculation and not by any intermediate calculation. The kind of the result is the same as that of the arguments, which must be an IEEE real kind.

22.7 Test sign

The elemental function `ieee_signbit` has been added to `ieee_arithmetic` to test the sign of a real as specified in the IEEE standard by the isSignMinus operation.

`ieee_signbit (x)` where `x` is type real and an IEEE kind, returns a default logical value that is true if and only if the sign bit of `x` is nonzero. Even when `x` is a signaling NaN, no exception is signaled by this function.

22.8 Conversion to integer type

The elemental function `ieee_int` has been added to `ieee_arithmetic` for conversion to integer type.

`ieee_int (a, round[, kind])` where `a` is of type real and IEEE kind, `round` is of type `ieee_round_type`, and `kind` is an integer constant expression specifying a valid integer kind, returns the value of `a` converted to an integer according to the rounding mode specified by `round`. If `kind` is absent, the result is default integer; otherwise the result is of type integer with kind type parameter `kind`, which must be a scalar integer constant expression. If the rounded value is not representable in the result kind, `ieee_invalid` is signaled and the result is processor dependent. The processor is required to consistently signal or consistently not signal `ieee_inexact` when the result is not exactly equal to `a`.

22.9 Remainder function

The arguments of the elemental function `ieee_rem` are now required to have the same radix.

22.10 Maximum and minimum values

The four elemental functions `ieee_max_num`, `ieee_max_num_mag`, `ieee_min_num`, and `ieee_min_num_mag` have been added for the IEEE operations of maxNum, maxNumMag, minNum, and minNumMag.

```
ieee_max_num (x, y)
ieee_max_num_mag (x, y)
ieee_min_num (x, y)
ieee_min_num_mag (x, y)
```

where `x` and `y` are real with the same kind type parameter, return either `x` or `y`. The result is `x` if `x>y`, `abs(x)>abs(y)`, `x<y`, or `abs(x)<abs(y)`, respectively. It is `y` if `x<y`, `abs(x)<abs(y)`, `x>y`, or `abs(x)>abs(y)`, respectively. If one of `x` and `y` is a quiet NaN, the result is the other. If one or both of `x` and `y` are signaling NaNs, `ieee_invalid` signals and the result is a NaN. Otherwise, which is returned is processor dependent for `ieee_max_num(x,y)` and `ieee_min_num(x,y)`; it is `ieee_max_num(x,y)` for `ieee_max_num_mag(x,y)`; and it is `ieee_min_num(x,y)` for `ieee_min_num_mag(x,y)`.

22.11 Adjacent machine numbers

The elemental functions `ieee_next_down` and `ieee_next_up` have been added for the IEEE operations of nextDown and nextUp.

`ieee_next_down (x)` where `x` is real returns the greatest value in the representation method of `x` that compares less than `x`, except that when `x` is equal to $-\infty$ the result has the value $-\infty$, and when `x` is a NaN the result is a NaN. If `x` is a signaling NaN, `ieee_invalid` signals; otherwise, no exception is signaled.

`ieee_next_up (x)` where `x` is real returns the least value in the representation method of `x` that compares greater than `x`, except that when `x` is equal to $+\infty$ the result has the value $+\infty$, and when `x` is a NaN the result is a NaN. If `x` is a signaling NaN, `ieee_invalid` signals; otherwise, no exception is signaled.

If `ieee_support_datatype(x)` has the value false, these functions must not be invoked. If `ieee_support_inf(x)` has the value false, `ieee_next_down(x)` must not be invoked when `x` has the value `-huge(x)` and `ieee_next_up(x)` must not be invoked when `x` has the value `huge(x)`.

22.12 Comparisons

Six elemental functions have been added for the IEEE operations of compareQuietEqual, compareQuietGreaterEqual, compareQuietGreater, compareQuietLessEqual, compareQuietLess, and compareQuietNotEqual. They are

```
ieee_quiet_eq (a, b)     ieee_quiet_le (a, b)
ieee_quiet_ge (a, b)     ieee_quiet_lt (a, b)
ieee_quiet_gt (a, b)     ieee_quiet_ne (a, b)
```

where `a` and `b` are real with the same kind type parameter; the result is true if `a` compares with `b` as equal, greater than or equal, greater than, less than or equal, less than, or not equal, respectively, and false otherwise. If `a` or `b` is a NaN, the result will be false. If `a` or `b` is a signaling NaN, `ieee_invalid` signals; otherwise, no exception is signaled.

Six elemental functions have been added for the IEEE operations of compareSignalingEqual, compareSignalingGreaterEqual, compareSignalingGreater, compareSignalingLessEqual, compareSignalingLess, and compareSignalingNotEqual. They are

```
ieee_signaling_eq (a, b)     ieee_signaling_le (a, b)
ieee_signaling_ge (a, b)     ieee_signaling_lt (a, b)
ieee_signaling_gt (a, b)     ieee_signaling_ne (a, b)
```

where `a` and `b` are real with the same kind type parameter; the result is true if `a` compares with `b` as equal, greater than or equal, greater than, less than or equal, less than, or not equal, respectively, and false otherwise. If `a` or `b` is a NaN, the result will be false and `ieee_invalid` signals; otherwise, no exception is signaled.

If x_1 and x_2 are of numeric types and the type of $x_1 + x_2$ is real, comparisons are made as follows:

$x_1 == x_2$	compareQuietEqual
$x_1 >= x_2$	compareSignalingGreaterEqual
$x_1 > x_2$	compareSignalingGreater
$x_1 <= x_2$	compareSignalingLessEqual
$x_1 < x_2$	compareSignalingLess
$x_1 /= x_2$	compareQuietNotEqual

If x_1 and x_2 are of numeric types and the type of $x_1 + x_2$ is complex, comparisons are made as follows:

$x_1 == x_2$	compareQuietEqual
$x_1 /= x_2$	compareQuietNotEqual

22.13 Hexadecimal significand input/output

Fortran 2018 supports the hexadecimal significand character sequence format specified by the IEEE standard for both input and output.

These character sequences begin with an optional sign followed by the hexadecimal indicator `0x` (or `0X`). This is followed by a hexadecimal significand, possibly containing a hexadecimal symbol (point or comma, the same as decimal), the exponent letter `p`, and the exponent, which is a power of two expressed as a decimal integer. For example, the decimal value 2.375 could be written as a hexadecimal-significand number as `0x2.6p+0` or `0x1.3p+1`. Note that the exponent is always required.

On input, hexadecimal significand character sequences are accepted as floating-point values by all edit descriptors for type real, as well as for list-directed and namelist input. For output, the new `ex` edit descriptor is used.

The `ex` edit descriptor has the forms `ex`*w*`.`*d* and `ex`*w*`.`*d*`e`*e*. For input, and for output of IEEE infinities and NaNs, this edit descriptor is treated identically to `f`*w*`.`*d*.

The field width *w* may be zero on output to let the processor choose the width. The output is unaffected by the scale factor and always has one digit before the hexadecimal symbol; if *d* is nonzero, the fraction contains *d* digits (rounded if necessary), otherwise the fraction contains the minimum number of digits required to represent the internal value exactly. Note that *d* is not permitted to be zero if the radix of the internal value is not a power of two (for example, if the internal value is a decimal floating-point number).

If *e* appears, it specifies the number of digits in the exponent. If the value of *e* is zero, or if *e* does not appear, the exponent part contains the minimum number of digits to represent the exponent.

The exponent for the value zero is always zero, but for a non-zero value the choice of the binary exponent is processor dependent. For example, the value 11.5 could be written as any of `0X1.7P+3`, `0X2.EP+2`, `0X5.CP+1`, or `0XB.8P+0`.

Table 22.1: Examples of hexadecimal-significand output.

Internal value	Edit descriptor	Possible output
1.375	`ex0.1`	`0X1.6P+0`
−15.625	`ex14.4e3`	`-0X1.F400P+003`
1 048 580.0	`ex0.0`	`0X1.00004P+20`

23. Minor Fortran 2018 features

23.1 Default accessibility for entities accessed from a module

If a module `a_mod` uses module `b_mod`, the default accessibility for entities it accesses from `b_mod` is the overall default accessibility for entities in `a_mod` (see Section 8.14). This is inconvenient when `a_mod` is default `public` but does not want to export very much if anything from `b_mod`, or when `a_mod` is default `private` but wants to export everything or nearly everything from `b_mod`.

Having to specify the non-default accessibility for most or all of `b_mod` can be tedious and error-prone. To overcome this, the default accessibility for entities accessed from a module can be controlled separately from the default accessibility of entities declared or defined locally. This is done by using the module name in a `public` or `private` statement; for example,

```
private b_mod
```

makes entities accessed by `use b_mod` have a default accessibility of `private` in `a_mod`, regardless of the overall default accessibility in `a_mod`. An explicit `public` or `private` specification for an entity used from `b_mod` is still permitted, and overrides or confirms the default.

A module name must not appear more than once in all the `public` and `private` statements in the using module.

23.2 Requiring explicit procedure declarations

A source of error in Fortran programs that is hard to detect is invoking a procedure that needs an explicit interface when no such interface is accessible. Similarly, invoking an external procedure when a module procedure was intended can also be difficult to find. This happens because using a name as a procedure implicitly declares it to be a procedure; to enable easier detection of such errors, this implicit declaration can now be disabled.

This is done by new syntax in the `implicit none` statement (Section 8.2); the general form of `implicit none` is now

`implicit none` *[* `(` *[implicit-none-spec-list]* `)` *]*

where each *implicit-none-spec* is `external` or `type`; each qualifier may appear at most once.

The appearance of `external` requires that the names of external and dummy procedures be explicitly declared to have the `external` attribute. This may be done by an `external`

Modern Fortran Explained, 2nd Edition. Michael Metcalf, John Reid, and Malcolm Cohen. Oxford University Press (2018). DOI 10.1093/oso/9780198811893.001.0001

statement, a `procedure` declaration statement (Section 14.1), an interface block, or the `external` attribute in a type declaration statement. Note that this still permits procedures to have an implicit interface, so long as they are explicitly declared to be procedures.

The appearance of `type` requires the types of all data entities to be explicitly declared, the same as `implicit none` without any qualification.

As before, `implicit none` affects the scoping unit and any contained scoping units. A contained scoping unit may have an `implicit` statement that overrides

```
implicit none (type)
```

for one or more initial letters of a name, but there is no mechanism for overriding

```
implicit none (external)
```

23.3 Using the properties of an object in its initialization

Fortran 2018 permits a property of a named constant or variable, such as an array bound or type parameter, to be used in its initialization expression. For example,

```
integer :: b = bit_size(b)
real, parameter :: rreps = sqrt(sqrt(epsilon(rreps)))
integer :: iota(10) = [ ( i, i = 1, size(iota,1) ) ]
character, parameter :: stars6*6 = repeat('*',6)
```

are now valid.

This only applies to the initialization expression, and when the property was determined by the declaration; that is, a property determined by the declaration cannot be used within the declaration itself, nor may a property declared by the initialization be used within that expression. For example, the following remain prohibited:

```
integer :: x(10, size(x, 1))
character(*),parameter :: c = repeat('c', len(c))
```

23.4 Generic procedures

23.4.1 More concise generic specification

Inside a derived-type definition, the `generic` statement (Section 15.8.2) can be used to declare type-bound generic interfaces. Fortran 2018 allows this statement to also be used to declare generic interfaces outside a derived-type definition. The syntax is similar to that of the type `generic` statement in a derived-type definition:

`generic` *[[, access-spec]* `::` *] generic-spec => specific-procedure-list*

where *access-spec* is `public` or `private`. This is more concise than using an interface block, reducing three statements (four if an *access-spec* is needed) to a single statement.

An interface block still needs to be used when a specific procedure needs to be declared with an interface body.

23.4.2 Rules for disambiguation

In Fortran 2018, if one procedure has more non-optional dummy procedures than the other has dummy procedures (including optional ones), they may be identified by the same generic name. It was anomalous that this means of disambiguation was absent from Fortran 2008 (Section 5.18).

23.5 Enhancements to stop and error stop

The stop code in a `stop` or `error stop` statement (Sections 5.3 and 17.14) is no longer required to be a constant expression. It can be any scalar expression of type integer or default character.

Output of the stop code and exception summary from a `stop` or `error stop` statement can be controlled with a `quiet=` specifier, for example

```
stop failure_message, quiet=no_messages
```

If the `quiet=` specifier is true, no stop code or exception summary is output. Any scalar logical expression may be used for the `quiet=` specifier.

23.6 New intrinsic procedures

23.6.1 Checking for unsafe conversions

The new elemental intrinsic function `out_of_range` has been added to test whether a real or integer value can be safely converted to a different real or integer type and kind.

`out_of_range(x,mold[,round])`

`x` is of type real or integer.

`mold` is of type real or integer.

`round` is of type logical and may be present only if `x` is of type real and `mold` is of type integer.

The arguments `mold` and `round` are required to be scalars. The result is of type default logical. It has the value true if and only if

- the value of `x` is an IEEE infinity or NaN, and `mold` is of type integer or is of type real and of a kind that does not support such a value;
- `mold` is of type integer, `round` is absent or present with the value false, and the integer with largest magnitude that lies between zero and `x` inclusive is not representable by objects with the type and kind of `mold`;
- `mold` is of type integer, `round` is present with the value true, and the integer nearest `x`, or the integer of greater magnitude if two integers are equally near to `x`, is not representable by objects with the type and kind of `mold`; or
- `mold` is of type real and the result of rounding the value of `x` to the extended model for the kind of `mold` has magnitude larger than that of the largest finite number with the same sign as `x` that is representable by objects with the type and kind of `mold`.

23.6.2 Generalized array reduction

A new transformational intrinsic function `reduce` for performing user-defined array reductions has been added. It is analogous to the collective subroutine `co_reduce` (Section 20.19).

`reduce (array, operation[, mask, identity, ordered])` or
`reduce (array, operation, dim[, mask, identity, ordered])`

`array` is an array of any type.

`operation` is a pure function that provides the binary operation for reducing the array. It is recommended that the operation be mathematically associative.[1] The function must have two scalar arguments and a scalar result, all non-polymorphic with the same type and type parameters as `array`. The arguments must not have the `allocatable`, `optional`, or `pointer` attributes; if one argument has the `asynchronous`, `target`, or `volatile` attribute, the other must have the same attribute.

`dim` is an integer scalar with value $1 \leq$ `dim` $\leq n$, where n is the rank of `array`.

`mask` is logical and conformable with `array`.

`identity` is a scalar of the type and type parameters of `array`.

`ordered` is a logical scalar.

The first form of this function returns a scalar of the type and type parameters of `array`. The result value is produced by taking the sequence of elements of `array` in array element order, and repeatedly combining adjacent values of the sequence by applying `operation` until only a single value is left. If `ordered` is present with the value true, the first value of the sequence is repeatedly combined with the next; otherwise any adjacent values could be combined (this is intended to permit efficient evaluation in parallel). If `mask` is present, the initial sequence consists only of those elements of `array` for which `mask` is true. If the initial sequence is empty, the result is `identity` if it is present; otherwise, error termination is initiated.

If `dim` appears and `array` is rank one, the result is the same as the first form. Otherwise, the result is an array with rank one less than `array`, and with the shape of array with dimension `dim` removed, that is,

`[ size(array,1)...size(array,dim-1), size(array,dim+1) ...size(array,`n`) ]`

Each element $(i_1, \ldots, i_{\texttt{dim}-1}, i_{\texttt{dim}+1}, \ldots, i_n)$ of the result has the value of applying `reduce` to `array`$(i_1, \ldots, i_{\texttt{dim}-1}, :, i_{\texttt{dim}+1}, \ldots, i_n)$.

23.6.3 Controlling the random number generator

Three problems have been found in Fortran 2008 with the use of the random number generator `random_number`:

i) Some processors always initialize the pseudorandom seed in the same way. Others purposely initialize it randomly.

[1] The operation need not be computationally associative – floating-point addition, for example.

ii) Each image might have its own pseudorandom seed or there might be a single seed and a single pseudorandom sequence that the images access in turns.

iii) Even with separate seeds, all images might or might not initialize the seeds to the same value.

For Fortran 2018, it was decided to require each image to have its own seed for the sake of efficiency when the number of images is very large. The intrinsic subroutine `random_init` has been added to control the initialization of the seed.

`random_init (repeatable, image_distinct)`

`repeatable` is a logical scalar of intent `in`. If it has the value true, the seed is set to a processor-dependent value that is the same each time `random_init` is called within a given execution environment from an image with the same image index in the initial team. If it has the value false, the seed is set to a processor-dependent, different value on each call.

`image_distinct` is a logical scalar of intent `in`. If it has the value true, the seed is set to a processor-dependent value that is distinct from the value that would be set by a call on another image to `random_init` with the same argument values. If it has the value false, the value to which the seed is set does not depend on which image calls `random_init`.

References to `random_init`, `random_number`, and `random_seed` update the seed on the executing image only.

23.7 Existing intrinsic procedures

23.7.1 Intrinsic function sign

In Fortran 2018, the arguments `a` and `b` to the intrinsic function `sign` (Section 9.3.2) can be of different kind. The kind of the result is that of argument `a`, which provides the magnitude.

23.7.2 Intrinsic functions extends_type_of and same_type_as

Because the dynamic type of a non-polymorphic pointer is always well defined, non-polymorphic pointer arguments to the intrinsic functions `extends_type_of` and `same_type_as` (Section 15.13) need not have defined pointer association status.

23.7.3 Simplification of calls of the intrinsic function cmplx

In a call of the intrinsic function `cmplx` (Section 9.3.1) with an argument of type complex, the keyword was needed for the `kind` argument. This requirement has been removed by regarding it as having two overloaded forms

```
cmplx(x[,kind])
cmplx(x[,y,kind])
```

23.7.4 Remove many argument dim restrictions

The function `count` has the form

```
count (mask[, dim, kind])
```

and a prohibition against the actual argument corresponding to `dim` being an optional dummy argument (see Section 9.13.2). This restriction is needed because whether `dim` is present may affect the rank of the result. In previous revisions of the standard it has been made unnecessary for the other functions of Section 9.13 by regarding each function as having two overloaded forms, for example

```
all(mask)
all(mask,dim)
```

For all those functions, the restriction on `dim` has been removed.

23.7.5 Kinds of arguments of intrinsic and IEEE procedures

Several of the intrinsic procedures and the IEEE module procedures require that their integer or logical arguments be of default kind. This restriction is completely unnecessary for type logical since all logical kinds can represent the same values, and unnecessarily restrictive for type integer since integers larger than default can represent all the values that a default integer can, and furthermore in many cases the values are inherently limited so that even a single-byte integer would suffice. These restrictions have been changed to improve consistency and usability. The arguments affected are as follows:

- Restriction changed to integer with a decimal exponent range of at least four:
  - the `values` argument of `date_and_time`;
  - the `cmdstat` argument of `execute_command_line`;
  - the `length` and `status` arguments of `get_command`, `get_command_argument`, and `get_environment_variable`.
- Restriction changed to integer with a decimal exponent range of at least nine:
  - the `exitstat` argument of `execute_command_line`.
- Restriction removed:
  - the `wait` argument of `execute_command_line`;
  - the `number` argument of `get_command_argument`;
  - the `dim` argument of `this_image`;
  - the `flag_value` arguments of `ieee_get_flag` and `ieee_set_flag`;
  - the `halting` arguments of `ieee_get_halting_mode` and `ieee_set_halting_mode`;
  - the `gradual` arguments of `ieee_get_underflow_mode` and `ieee_set_underflow_mode`.

23.7.6 Intrinsic subroutines that access the computing environment

An extra optional argument `errmsg` has been added to each of the intrinsic subroutines `get_command`, `get_command_argument`, and `get_environment_variable` (Section 9.19). It is a scalar intent `inout` argument of type default character and returns an error message if an error occurs. In the case of a warning situation that would assign -1 to the argument `status`, `errmsg` is left unchanged. This change brings these procedures into line with the ability to retrieve error messages for I/O statements via the `iomsg=` specifier, image control statements via the `errmsg=` specifier, and invocations of `execute_command_line` via the `cmdmsg` argument.

The effects of invoking the intrinsic procedures `command_argument_count`, `get_command`, `get_command_argument`, and `get_environment_variable` on images other than image one of the initial team are now as on image one. This is important with the addition of teams, because a team need not include image one. In the case of `get_environment_variable` it is processor dependent whether an environment variable that exists on an image also exists on another image, and if it does exist on both images, whether the values are the same.

23.8 Use of non-standard features

Fortran has long required the processor to have the capability to detect and report the use of a non-standard intrinsic procedure. In Fortran 2018, the processor is further required to have the capability to detect and report the use of a non-standard intrinsic module, and the use of a non-standard procedure of a standard intrinsic module. This includes a non-standard use of a standard procedure of a standard intrinsic module; for example, if a processor permits the use of `c_loc` on a polymorphic variable (forbidden by the standard), it must be capable of reporting such usage.

23.9 Kind of the do variable in implied-do loops

In Fortran 2008, for a `do concurrent` construct,[2] the index variables have the scope of the construct, and can have their type and kind explicitly specified, for example

```
do concurrent (integer(int64)::i=1:n, j=1:m, i/=j)
```

where `int64` is the named constant from the intrinsic module `iso_fortran_env` (Section 9.24.3).

The `do` variables in implied-`do` loops in array constructors (Section 7.16) and `data` statements (Section 8.5.2) similarly have limited scope (in this case to the implied-`do` loop), but there was no way to specify their type and kind within the loop, necessitating the declaration of a variable in the scoping unit when the implicit type and kind were not as desired (for example, when implicit typing has been disabled by an `implicit none` statement).

Fortran 2018 provides syntax for explicitly specifying the type and kind of such `do` variables, for example

[2] Also for a `forall` construct, or `forall` statement, Appendix B.3.1.

```
[ (a(i,i), integer(int64) :: i=1,n) ]
data ((a(i,j), integer(int64) :: i=1,5,2), j=1,5) /15*0./
```

In these cases, the type must be `integer` and applies only to the `do` variable of that implied-`do` loop.

Note that this feature is not available for an implied-`do` in an input/output list, as the `do` variable in that case does not have limited scope but is the variable in the scoping unit.

23.10 Improving do concurrent performance

The rules on the use of a variable within a `do concurrent` construct (Section 7.17) allow the processor to execute the iterations in any order; however, it can only execute them in parallel if it can discover whether every variable used in the construct should be local to each iteration (is assigned a value before further use in that iteration) or shared between all iterations (is assigned a value by at most one iteration). While this is straightforward for many simple examples, it can be impossible to discover this at compile time if there is a conditional assignment to the variable. In such cases there is no choice but for sequential execution, that is, little or no performance benefit was obtained by using `do concurrent` over a normal `do` loop.

To address this problem, the syntax of the `do concurrent` statement has been extended to include specifications of the **locality** of variables as follows:

`do` *[,]* `concurrent (` *[type-spec* `::` *] index-spec-list [, mask-expr])* *[locality-spec]* ...

where *locality-spec* is one of

`default (none)`
`local (`*variable-name-list*`)`
`local_init (`*variable-name-list*`)`
`shared (`*variable-name-list*`)`

Each variable named must exist in the scope of the `do concurrent` statement, and must not be the same as any *index-variable-name* of the statement.

A variable that does not appear in a `local`, `local_init`, or `shared` clause has *unspecified locality* and the rules of Fortran 2008 apply to it. The `default (none)` clause specifies that no variable has unspecified locality; if a variable used in the loop did not appear in any *locality* clause, a compile-time error will be produced.

The `local` and `local_init` clauses declare that each named variable is a *construct entity* of the `do concurrent`, and that the outside variable with the same name is inaccessible. The construct entity has the same type, type parameters, and rank as the outside variable, and has the `asynchronous`, `contiguous`, `pointer`, `target`, or `volatile` attribute if the outside variable has the attribute. Even if the outside variable has the `bind`, `intent`, `protected`, `save`, or `value` attribute, the construct entity does not have the attribute. The outside variable is not permitted to be allocatable, intent `in`, optional, a coarray, an assumed-size array, a non-pointer polymorphic dummy argument, of finalizable type, or be a variable that is not permitted to appear in a context that could change its value (for example, a `protected` variable outside its module).

At the beginning of each iteration, a variable that is `local` is undefined, and if a pointer, has undefined association status, whereas a `local_init` variable begins each iteration with the

value or pointer association status of the outside variable. Changes to `local` and `local_init` variables do not affect the outside variable.

The `shared` clause declares that each named variable is not a construct entity, but is the same variable as outside the construct. If it is defined or becomes undefined during any iteration, it must not be referenced, defined, or become undefined during any other iteration. If it is allocated, deallocated, nullified, or pointer-assigned during an iteration, it must not have its allocation or association status, dynamic type, array bounds, shape, or a deferred type parameter value inquired about in any other iteration. A discontiguous array with `shared` locality must not be supplied as an actual argument corresponding to a contiguous intent `inout` dummy argument.

A simple example of the use of locality clauses is shown in Figure 23.1.

Figure 23.1 Simple example of locality clauses in `do concurrent`.

```
real a(:), b(:), x
⋮
do concurrent (i=1:size(a)) local (x) shared (a, b)
   if (a(i)>0) then
      x = sqrt (a(i))
      a(i) = a(i) - x**2
   end if
   b(i) = b(i) - a(i)
end do
⋮
```

23.11 Control of host association

The `import` statement can be used in a contained subprogram or block construct to control host association (Section 5.11.1). In addition to the old form

`import` *[[*`::`*] import-name-list]*

it has the new forms

```
import, only:  import-name-list
import, none
import, all
```

If one `import` statement in a scoping unit is an `import,only` statement, they must all be, and only the entities listed become accessible by host association.

If an `import,none` statement appears in a scoping unit, no entities are accessible by host association and it must be the only `import` statement in the scoping unit; `import,none` is not permitted in the specification part of a submodule except within an interface body.

If an `import,all` statement appears in a scoping unit, all entities of the host are accessible by host association and it must be the only `import` statement in the scoping unit.

Each *import-name* must be the name of a host entity and must not appear in a context that makes the host entity inaccessible, such as being declared as a local entity. For `import,all` no entity of the host may be inaccessible for this reason. This restriction does not apply to the simple form of `import`.

23.12 Intent in requirements and the value attribute

In Fortran 2008, nearly everywhere that intent `in` was formerly required, for example for arguments of pure functions, the `value` attribute is permitted instead. This is because a dummy argument with the `value` attribute is effectively a local variable, so changing it has no effect on the actual argument.

There were two places where intent `in` was still required even for a `value` argument: an argument of a function specified as a defined operation (Section 3.9), and the second argument of a subroutine specified as a defined assignment (Section 3.10). In Fortran 2018 this oversight has been corrected, so that if the argument has the `value` attribute, intent `in` is no longer required.

23.13 Pure procedures

Execution of an `error stop` statement causes execution to cease as soon as possible on all images, so there is no need to disallow it in a `pure` procedure. It is now allowed and gives the programmer the opportunity to provide an explanation in the *stop-code*.

The standard procedures in the intrinsic module `iso_c_binding`, other than `c_f_pointer`, are now pure.

A dummy argument of a pure function that has the `value` attribute, and is not intent `in`, may have its value changed, for example by an assignment statement. Such a variable is effectively a local variable, so changing it cannot cause any harmful side effect.

23.14 Recursive and non-recursive procedures

In Fortran 2018, procedures, apart from functions whose result is of type `character` with an asterisk character length,[3] are recursive by default and the keyword `non_recursive` has been added to allow a procedure to be specified as non-recursive. It can be used wherever `recursive` can be used. Using a non-recursive procedure recursively is likely to produce invalid results on a system that does not detect such use, so it is safer to require the keyword `non_recursive` for the case where the user is sure that this is all that is wanted.

Similarly, the restriction against elemental recursion was intended to make elemental procedures easier to implement and optimize, but has proven to be unnecessary for optimization and inconvenient for the programmer. It has been removed.

[3] An obsolescent feature, see Section B.1.6 – such functions cannot be recursive.

23.15 Input/output

23.15.1 More minimal field width editing

It was anomalous that the field width *w* was allowed to be given as zero for some output `g` editing, but not for `d`, `e`, `es`, `en`, and all `g` output editing (see Section 11.3). This has been corrected. If the value is zero, the processor selects the smallest positive actual field width that does not result in a field filled with asterisks. A zero value for the field width is not permitted for input.

The `g0.`*d* edit descriptor can be used to specify the output of integer, logical, and character data. It follows the rules for the `i0`, `l1`, and `a` edit descriptors, respectively.

Similarly, `0` may be specified as the exponent width parameter *e* in the `e`*w.d*`e`*e*, `en`*w.d*`e`*e*, `es`*w.d*`e`*e*, and `g`*w.d*`e`*e* edit descriptors. (But note that `g`*w.d*`e`*e* does not permit *w* to be zero, unlike the others.) The effect of `e0` is that the processor uses the minimum number of digits required to represent the exponent value, that is, without leading zeros.

For example, writing the value −125.0 with `es0.3e0` produces the output `-1.250E+2`.

23.15.2 Recovering from input format errors

In previous Fortran standards, data processed by logical or numeric editing during execution of a formatted input statement was required to have the correct form. This made recovery from input format errors unreliable, or at least non-portable, as in such cases the program could produce unexpected behaviour, or crash even with an `iostat=` specifier.

Fortran 2018 specifies that if the form of the input data is not of the standard form (for example, if it contains an invalid character for numeric editing), and is not otherwise acceptable to the processor, an input/output error condition occurs. This error condition can be caught with an `iostat=` or `err=` specifier; in the absence of such a specifier, error termination will occur (instead of undefined behaviour).

23.15.3 Advancing input with size=

The prohibition against `size=` (Section 10.11) appearing with advancing input has been removed.

23.15.4 Precision of stat= variables

It is recommended that any `stat=` variable, for instance in an `allocate` statement (Section 6.5), should have a decimal exponent range of at least four to ensure that the error code is representable in the variable.

23.15.5 Connect a file to more than one unit

It can be convenient to connect a file to more than one unit at a time. For example, the two units might be reading different parts of the same sequential file. While the prohibition

against connecting a file to more than one unit at a time (Section 12.1) has been removed, there is no requirement on the processor to provide this functionality. For a given file and action choice, whether or not it is available is processor dependent; however, most operating systems will permit a file to be open with `action='read'` on more than one unit at the same time.

23.15.6 Enhancements to inquire

If a `recl=` inquiry is made in an `inquire` statement in Fortran 2008 (Section 12.6) and there is no connection or the connection is for stream access, the variable becomes undefined. In Fortran 2018, it is assigned the value -1 if there is no connection and the value -2 if the connection is for stream access.

If a `pos=` or `size=` inquiry is made in an `inquire` statement and there are pending data transfer operations for the specified unit, the value assigned is computed as if all the pending data transfers had already completed.

23.15.7 Asynchronous communication

The `asynchronous` attribute has been extended from I/O to apply to communication performed by means other than Fortran. The main application is for non-blocking calls of `MPI_Irecv` and `MPI_Isend`. A call of `MPI_Irecv` is very like asynchronous input – data is put in the buffer array while execution continues; a call of `MPI_Isend` is very like asynchronous output – data is copied from the buffer array while execution continues. For both, a call of `MPI_Wait` plays the role of `wait` for asynchronous I/O.

The standard does not limit asynchronous communication to these MPI functions. Instead, it talks in general of procedures that initiate input or output asynchronous communication or complete it. Whether a procedure has such a property is processor dependent.

The rules for input and output asynchronous communication are exactly the same as those for asynchronous input and output, respectively.

23.16 Assumed rank

23.16.1 Assumed-rank objects

The concept of **assumed rank** has been added to make writing procedures that can handle any rank easier. It is particularly useful in combination with C functions that have been written to handle arguments of any rank (Section 21.6.6).

A dummy argument that is not a coarray and does not have the `value` attribute may be declared to have assumed rank with the syntax `(..)`. For example, the procedure

```
subroutine scale(a) bind(c)
   real a(..)
   ⋮
end subroutine scale
```

may be called with an array of any rank or even a scalar as an actual argument. The dummy argument assumes the rank and shape of the actual argument; an assumed-size actual argument has no shape, so in this case the extents are assumed (except for the final dimension, which has no extent).

The interface of a procedure with an assumed-rank dummy argument is required to be explicit.

Because an assumed-rank dummy argument may be associated with an assumed-size dummy argument, when it has intent `out` the same restrictions apply as for assumed size: that is, it must not be polymorphic, finalizable, of a type with an allocatable ultimate component, or of a type for which default initialization is specified.

A new intrinsic inquiry function is provided:

`rank (a)` returns a default integer scalar whose value is the rank of `a`, which may be a scalar or array of any type.

Because the rank of an assumed-rank object is not known at compile time, its direct uses in a program are limited:

- it may be an actual argument corresponding to an assumed-rank dummy argument;
- it may be the argument of `c_loc` or `c_sizeof`[4] from the intrinsic module `iso_c_binding`;
- it may be the first argument of an intrinsic inquiry function.

An assumed-rank object must not appear in any other context, including in a variable designator. For example, an assumed-rank complex variable `arc` may be passed to a complex assumed-rank dummy, but its real or imaginary parts `arc%re` and `arc%im` are not permitted anywhere, not even as an actual argument to a real assumed-rank dummy.

Because a scalar object has rank zero, when an assumed-rank object `arx` is associated with a scalar, `shape(arx)` will return a zero-sized array, and `size(arx)` will return the value 1. Note that because the `dim` argument to the `size` intrinsic is required to satisfy $1 \leq \texttt{dim} \leq \texttt{rank(array)}$, it must not be used on an assumed-rank object when it is associated with a scalar.

When an assumed-rank object `arx` is associated with an assumed-size array, because it has no final extent, `shape(arx)` will return an array with a final extent value of -1. Similarly, `size(arx, rank(arx))` will return the value -1. However, care should be taken with `size(arx)` with no `dim` argument; this could return a negative value, but due to the possibility of integer overflow, could return any value. For example, when associated with an assumed-size array declared as `a(65536,65536,65536,0)`, `size(arx)` is likely either to return the value zero, or produce a run-time error.

Similar considerations apply to the intrinsic functions `lbound` and `ubound`. When associated with a scalar, the `dim` form must not be used and the non-`dim` form returns a zero-sized array. When associated with an assumed-size array, `ubound` returns a value for the final dimension that is two less than the lower bound; this maintains the identity

$$\texttt{size} = \texttt{ubound} - \texttt{lbound} + 1$$

[4] When used in `c_sizeof`, it must not be associated with an assumed-size array.

For the purposes of generic resolution, an assumed-rank dummy argument is considered not to be distinguishable from any other rank, and therefore the generic interface

```
interface g
   subroutine scalar(x)
      real x
   end subroutine
   subroutine sassumed(x)
      real x(..)
   end subroutine
end interface
```

is not a valid generic interface.

23.16.2 The select rank construct

To execute alternative code depending on the actual rank of an assumed-rank object, the `select rank` construct is provided. It has the form

[name `:` *]* `select rank (` *[associate-name =>* *]* *selector)*
[rank-case-stmt [name]
block] ...
`end select` *[name]*

where *selector* is the name of an assumed-rank object and each *rank-case-stmt* is one of

`rank (` *expr* `)`
`rank (*)`
`rank default`

Each *expr* must be a scalar integer constant expression, and each must have a distinct non-negative value. The value in a `rank (` *expr* `)` statement is permitted to exceed the maximum possible array rank, in which case its block can never be executed. This is to enhance portability of programs between processors with different maximum array rank.

As for other constructs, the leading and trailing statements must either both be unnamed or both bear the same name; a *rank-case-stmt* within it may be named only if the `select rank` statement is named and bears the same name.

If *associate-name* is omitted, the associate name is the same as the *selector*. Within the *block* of a `rank (` *expr* `)` statement, the associate name has the rank specified by *expr* with bounds the same as returned by the intrinsic functions `lbound` and `ubound` for the *selector*. Within the *block* of a `rank (*)` statement, the associate name is assumed size with rank one and lower bound one. Within the *block* of a `rank default` statement, the associate name is assumed rank, the same as the *selector*.

Executing a `select rank` construct selects the block to be executed according to the rank of the *selector* and whether it is assumed size. If it is not assumed size and a `rank (` *expr* `)` statement appears with the same rank, its block is executed; if it is assumed size and a `rank (*)` statement appears, its block is executed; otherwise, if a `rank default` statement appears, its block is executed.

For example, the function in Figure 23.2 computes a function of an object depending on its rank.

Figure 23.2 Simple example of assumed rank

```
double precision function f(x)
   double precision, intent(in) :: x(..)
   select rank (x)
   rank (*)
      error stop 'Function not defined on assumed-size arrays'
   rank (0)
      f = abs(x)
   rank (1)
      f = sum(abs(x))
   rank (2)
      f = sqrt(sum(x**2))
   rank (3)
      f = sum(abs(x)**3)**(1/3.0d0)
   rank (4)
      f = sum(x**4)**(0.25d0)
   rank default
      error stop 'Function not supported for rank>4 (unstable)'
end function
```

A. Deprecated features

A.1 Introduction

This appendix begins by describing features that became redundant with Fortran 95 and whose use we have since deprecated. Descriptions of newer deprecated features follow in Section A.9. They might become obsolescent in a future revision, but this is a decision that can be made only within the standardization process. We note that this decision to group certain features into an appendix and to deprecate their use is ours alone, and does not have the actual or implied approval of either WG5 or J3.

Each description mentions how the feature concerned may be effectively replaced by a newer feature.

A.2 Storage association

Storage units are the fixed units of physical storage allocated to certain data. There is a storage unit called **numeric** for any non-pointer scalar of the default real, default integer, and default logical types, and a storage unit called **character** for any non-pointer scalar of type default character and character length 1. Non-pointer scalars of type default complex or double precision real (Appendix A.6) occupy two contiguous numeric storage units. Non-pointer scalars of type default character and length *len* occupy *len* contiguous character storage units.

As well as numeric and character storage units, there are a large number of **unspecified** storage units. A non-pointer scalar object of type non-default integer, real other than default or double precision, non-default logical, non-default complex, or non-default character of any particular length occupies a single unspecified storage unit that is different for each case. A data object with the `pointer` attribute has an unspecified storage unit, different from that of any non-pointer object and different for each combination of type, type parameters, and rank. The standard makes no statement about the relative sizes of all these storage units and permits storage association to take place only between objects with the same category of storage unit.

A non-pointer array occupies a sequence of contiguous storage sequences, one for each element, in array element order.

Objects of derived type have no storage association, each occupying an unspecified storage unit that is different in each case, except where a given type with at least one component contains a `sequence` statement making it a **sequence type**, for example:

```
type storage
   sequence
   integer i          ! First numeric storage unit;
   real a(0:999)      ! subsequent 1000 numeric storage units.
end type storage
```

Should any other derived types appear in such a definition, they too must be sequence types. The type is not allowed to have type parameters (Section 13.2) or type-bound procedures (Section 15.8).

A sequence type is a **numeric sequence type** if no component is a pointer or allocatable, and each component is of type default integer, default real, double precision real, default complex, or default logical. A component may also be of a previously defined numeric sequence type. This implies that the ultimate components occupy numeric storage units and the type itself has **numeric storage association**. Similarly, a sequence type is a **character sequence type** if no component is a pointer or allocatable, and each component is of type default character or a previously defined character sequence type. Such a type has **character storage association**.

A scalar of numeric or character sequence type occupies a storage sequence that consists of the concatenation of the storage sequences of its components. A scalar of any other sequence type occupies a single unspecified storage unit that is unique for each type.

A `private` statement may be added to a sequence derived-type definition, making its components private. The `private` and `sequence` statements may be interchanged but must be the second and third statements of the derived-type definition.

Two derived-type definitions in different scoping units define the same data type if they have the same name,[1] both have the `sequence` attribute, and they have components that are not `private` and agree in order, name, and attributes. Argument association may occur between objects of types that are the same in this way, and this is known as sequence association. Such a practice is prone to error and offers no advantage over having a single definition in a module that is accessed by use association.

A sequence type is permitted to have an allocatable component, which permits independent declarations of the same type in different scopes, but such a type, like a pointer, has an unspecified storage unit.

Use of equivalence statements (Section B.3.2) and `common` blocks (Section B.3.3) are the main way in which storage association occurs.

Should that be required, the intrinsic module `iso_fortran_env` provides information about the Fortran storage environment, in the form of named constants as follows:

`character_storage_size` The size in bits of a character storage unit.

`numeric_storage_size` The size in bits of a numeric storage unit.

All use of storage association is error prone and we do not recommend it.

A.3 Alternative form of relational operator

The relational operators have alternative and redundant forms

[1] If one or both types have been accessed by use association and renamed, it is the original names that must agree.

`.lt.`	for	`<`	less than
`.le.`	for	`<=`	less than or equal
`.eq.`	for	`=`	equal
`.ne.`	for	`/=`	not equal
`.gt.`	for	`>`	greater than
`.ge.`	for	`>=`	greater than or equal

Corresponding operator forms are considered to be the same operator; for example, providing a generic interface for `.ne.` also provides it for `/=`.

A.4 The include line

It is sometimes useful to be able to include source text from somewhere else into the source stream presented to the compiler. This facility is possible using an `include` line:

`include` *char-literal-constant*

where *char-literal-constant* must not have a kind parameter that is a named constant. This line is not a Fortran statement and must appear as a single source line where a statement may occur. It will be replaced by material in a processor-dependent way determined by the character string *char-literal-constant*. The included text may itself contain `include` lines, which are similarly replaced. An `include` line must not reference itself, directly or indirectly. When an `include` line is resolved, the first included line must not be a continuation line and the last line must not be continued. An `include` line may have a trailing comment, but may not be labelled nor, when expanded, may it contain incomplete statements.

The `include` line was available as an extension to many Fortran 77 systems and was often used to ensure that every occurrence of global data in a `common` block was identical. In modern Fortran the same effect is better achieved by placing global data in a module (Section 5.5). This cannot lead to accidental declarations of local variables in each procedure.

This feature is useful when identical executable statements are needed for more than one type, for example in a set of procedures for sorting data values of various types. The executable statements can be maintained in an include file that is referenced inside each instance of the sort procedure.

A.5 The do while statement

In Section 4.4 a form of the `do` construct was described that may be written as

```
do
   if (scalar-logical-expr) exit
   ⋮
end do
```

An alternative, but redundant, form of this is its representation using a `do while` statement:

[name:] `do` *[label] [,]* `while (.not.`*scalar-logical-expr*`)`

We prefer the form that uses the `exit` statement because this can be placed anywhere in the loop, whereas the `do while` statement always performs its test at the loop start. If *scalar-logical-expr* becomes false in the middle of the loop, the rest of the loop is still executed. Potential optimization penalties that the use of `do while` entails are fully described in Chapter 10 of *Optimizing Supercompilers for Supercomputers*, M. Wolfe (Pitman, 1989).

A.6 Double precision real

Another *type* that may be used in a type declaration, `function`, `implicit`, or component declaration statement is `double precision` which specifies double precision real. The precision is greater than that of default real.

Literal constants written with the exponent letter `d` (or `D`) are of type double precision real by default; no kind parameter may be specified if this exponent letter is used. Thus, `1d0` is of type double precision real. If `dp` is an integer named constant with the value `kind(1d0)`, `double precision` is synonymous with `real(kind=dp)`.

There is a `d` (or `D`) edit descriptor that was originally intended for double precision quantities, but now is identical to the `e` edit descriptor except that the output form may have a `D` instead of an `E` as its exponent letter. A double precision real literal constant, with exponent letter `d`, is acceptable on input whenever any other real literal constant is acceptable.

There are two elemental intrinsic functions which were not described in Chapter 9 because they have a result of type double precision real:

`dble (a)` for `a` of type integer, real, or complex returns the double precision real value `real(a, kind(0d0))`.

`dprod (x, y)` returns the product `x*y` for `x` and `y` of type default real as a double precision real result.

The double precision real data type has been replaced by the real type of kind `kind(0.d0)`.

A.7 The dimension, codimension, and parameter statements

To declare entities we normally use type specifications. However, if all the entities involved are arrays, they may be declared *without* type specifications in a `dimension` statement:

```
dimension i(10), b(50,50), c(n,m) ! n and m are integer dummy
                                  ! arguments or named constants
```

The general form is

`dimension` *[* `::` *]* *array-name* `(`*array-spec*`)` *[* `,` *array-name* `(`*array-spec*`)` *]* ...

Here, the type may either be specified in a type declaration statement such as

```
integer i
```

that does not specify the dimension information, or may be declared implicitly. Our view is that neither of these is sound practice; the type declaration statement looks like a declaration of a scalar and we explained in Section 8.2 that we regard implicit typing as dangerous. Therefore, the use of the `dimension` statement is not recommended.

The same remark applies to the `codimension` statement for declaring coarrays (Chapter 17) with the syntax

`codimension` *[* `::` *]* *coarray-decl-list*

where each *coarray-decl* is

coarray-name [`(`*array-spec*`)` *]* `[` *coarray-spec* `]`

An alternative way to specify a named constant is by the `parameter` statement. It has the general form

`parameter (` *named-constant-definition-list* `)`

where each *named-constant-definition* is

constant-name `=` *constant-expr*

Each constant named must either have been typed in a previous type declaration statement in the scoping unit, or take its type from the first letter of its name according to the implicit typing rule of the scoping unit. In the case of implicit typing, an appearance of the named constant in a subsequent type declaration statement in the scoping unit must confirm the type and type parameters, and there must not be an `implicit` statement for the letter subsequently in the scoping unit. Similarly, the shape must have been specified previously or be scalar. Each named constant in the list is defined with the value of the corresponding expression according to the rules of intrinsic assignment.

An example using implicit typing and a constant expression including a named constant that is defined in the same statement is

```
implicit integer (a, p)
parameter (apple = 3, pear = apple**2)
```

For the same reasons as for `dimension`, we recommend avoiding the `parameter` statement.

A.8 Non-default mapping for implicit typing

The default for implicit typing (Section 8.2) is that entities whose names begin with one of the letters `i`, `j`, ..., `n` are of type default integer, and variables beginning with the letters `a`, `b`, ..., `h` or `o`, `p`, ..., `z` are of type default real. If implicit typing with a different rule is desired in a given scoping unit, the `implicit` statement may be employed. This changes the mapping between the letters and the types with statements such as

```
implicit integer (a-h)
implicit real(selected_real_kind(10)) (r,s)
implicit type(entry) (u,x-z)
```

The letters are specified as a list in which a set of adjacent letters in the alphabet may be abbreviated, as in `a-h`. No letter may appear twice in the implicit statements of a scoping unit and, if there is an `implicit none` statement, there must be no other implicit statement in the scoping unit. For a letter not included in the implicit statements, the mapping between the letter and a type is the default mapping.

In the case of a scoping unit other than a program unit or an interface block, for example a module subprogram, the default mapping for each letter in an inner scoping unit is the mapping for the letter in the immediate host. If the host contains an `implicit none` statement, the default mapping is null and the effect may be that implicit typing is available

for some letters, because of an additional `implicit` statement in the inner scope, but not for all of them. The mapping may be to a derived type even when that type is not otherwise accessible in the inner scoping unit because of a declaration there of another entity with the same name.

Implicit typing does not apply to an entity accessed by use or host association because its type is the same as in the module or the host. Figure A.1 provides a comprehensive illustration of the rules of implicit typing.

The general form of the `implicit` statement is

```
implicit none
```

or

`implicit` *type* `(`*letter-spec-list*`)` *[* `,` *type* `(`*letter-spec-list*`)` *]* ...

where *type* specifies the type and type parameters (Section 8.17) and each *letter-spec* is

letter [`-` *letter]*

The `implicit` statement may be used for a derived type. For example, given access to the type

```
type posn
   real    :: x, y
   integer :: z
end type posn
```

and given the statement

```
implicit type(posn) (a,b), integer (c-z)
```

variables beginning with the letters `a` and `b` are implicitly typed `posn` and variables beginning with the letters `c`, `d`, ..., `z` are implicitly typed integer.

An `implicit none` statement may be preceded within a scoping unit only by `use` (and `format`) statements, and other `implicit` statements may be preceded only by `use`, `parameter`, and `format` statements. We recommend that each `implicit none` statement be at the start of the specifications, immediately following any `use` statements.

A.9 Fortran 2008 deprecated features

In this section we describe features that were new in Fortran 2008, but are considered by us to be redundant.

A.9.1 The sync memory statement, and atomic_define and atomic_ref

The execution of a `sync memory` statement defines a boundary on an image between two segments, each of which can be ordered in some user-defined way with respect to segments on other images. Unlike the other image control statements it does not have any in-built synchronization effect. In case there is some user-defined ordering between images, the compiler will probably avoid optimizations involving moving statements across the `sync memory` statement and will ensure that any changed data that the image holds in temporary

Figure A.1 Illustration of the rules of implicit typing.

```
module example_mod
   implicit none
   ⋮
   interface
      function fun(i)     ! i is implicitly
         integer :: fun   ! declared integer.
      end function fun
   end interface
contains
   function jfun(j)       ! All data entities must
      integer :: jfun, j  ! be declared explicitly.
      ⋮
   end function jfun
end module example_mod
subroutine sub
   implicit complex (c)
   c = (3.0,2.0)          ! c is implicitly declared complex
   ⋮
contains
   subroutine sub1
      implicit integer (a,c)
      c = (0.0,0.0) ! c is host associated and of type complex
      z = 1.0       ! z is implicitly declared real.
      a = 2         ! a is implicitly declared integer.
      cc = 1.0      ! cc is implicitly declared integer.
      ⋮
   end subroutine sub1
   subroutine sub2
      z = 2.0       ! z is implicitly declared real and is
                    ! different from the variable z in sub1.
      ⋮
   end subroutine sub2
   subroutine sub3
      use example_mod     ! Access the integer function fun.
      q = fun(k)          ! q is implicitly declared real and
                          ! k is implicitly declared integer.
      ⋮
   end subroutine sub3
end subroutine sub
```

memory, such as cache or registers, or even packets in transit between images, are made visible to other images. Also, any data from other images that are held in temporary memory will be treated as undefined until it is reloaded from its host image.

We see the construction of reliable and portable code in this way as very difficult – it is all too easy to introduce subtle bugs that manifest themselves only occasionally.

One way to effect user-defined ordering between images is by employing the atomic subroutines `atomic_define` and `atomic_ref`:

`call atomic_define (atom, value)` defines `atom` atomically with the value `value`.

`atom` has intent `out` and is a scalar coarray or coindexed object of type `integer(atomic_int_kind)` or `logical(atomic_logical_kind)`; the kinds `atomic_int_kind` and `atomic_logical_kind` are named constants in the standard intrinsic module `iso_fortran_env`.

`value` has intent `in` and is a scalar of the same type as `atom`. Its value must be representable in the kind of `atom`.

`call atomic_ref (value, atom)` gets the value of `atom` atomically and assigns it to `value`.

`value` has intent `out` and is a scalar of the same type as `atom`.

`atom` has intent `in` and is a scalar coarray or coindexed object. It has type `integer(atomic_int_kind)` or `logical(atomic_logical_kind)`.

In Fortran 2018 an optional `stat` argument has been added; this is a scalar integer intent `out` argument with a decimal exponent range of at least four, and must not be a coindexed object. If present, it is assigned the value zero if no error condition occurs. If an error condition occurs, the program will be terminated if `stat` is not present, otherwise a positive error code will be assigned to `stat` and the argument that would have been defined becomes undefined.

As an example of user-defined ordering, consider the code in Figure A.2, which is executed on images `p` and `q`. The do loop is known as a spin-wait loop. Once image `q` starts executing it, it will continue until it finds the value `.false.` for `val`. The `atomic_ref` call ensures that the value is refreshed on each loop execution. The effect is that the segment on image `p` ahead of the first `sync memory` statement precedes the segment on image `q` that follows the second `sync memory` statement. The normative text of the standard does not specify how resources should be distributed between images, but a note expects that the sharing should be equitable. It is therefore just possible that a conforming implementation might give all its resources to the spin loop while doing nothing on image `p`, causing the program to hang.

Note that the segment in which `locked[q]` is altered is unordered with respect to the segment in which it is referenced. This is permissible by the rules in the penultimate paragraph of Section 17.13.1.

Given the atomic subroutines and the `sync memory` statement, customized synchronizations can be programmed in Fortran as procedures, but it may be difficult for the programmer to ensure that they will work correctly on all implementations.

Figure A.2 Spin-wait loop.

```
use, intrinsic :: iso_fortran_env
logical(atomic_logical_kind) :: locked[*] = .true.
logical :: val
integer :: iam, p, q
⋮
iam = this_image()
if (iam == p) then
   sync memory
   call atomic_define(locked[q], .false.)
   ! Has the effect of locked[q]=.false.
else if (iam == q) then
   val = .true.
   ! Spin until val is false
   do while (val)
      call atomic_ref(val, locked)
      ! Has the effect of val=locked
   end do
   sync memory
end if
```

All of the image control statements except `critical`, `end critical`, `lock`, and `unlock` include the effect of executing a `sync memory` statement.

A.9.2 Components of type c_ptr or c_funptr

A coarray is permitted to have a component of type `c_ptr` or `c_funptr`, but a coindexed object is not permitted to be of either of these types because it is almost certain to involve a remote reference. Furthermore, intrinsic assignment for either of these types causes the variable to become undefined unless the variable and expression are on the same image. It is very hard to see good uses for this feature.

A.9.3 Type declarations

The `type` keyword can be used with an intrinsic type specification instead of a derived type specification, and this declares the entities to be of that intrinsic type. For example,

```
type(complex(kind(0d0))) :: a, b, c
```

declares `a`, `b`, and `c` to be of intrinsic type `complex` with kind type parameter equal to `kind(0d0)`, that is double precision complex. This syntax is completely redundant and the example is equivalent to

```
complex(kind(0d0)) :: a, b, c
```

This feature was added for consistency with the `type is` statement in the `select type` construct: in that statement an intrinsic type is specified by its keyword, but a derived type is specified simply by its type name without the `type` keyword (or the concomitant parentheses).

We consider that this feature adds nothing to the language; furthermore, it might confuse a reader into thinking that an intrinsic type is really a derived type, so we do not recommend its use.

A.9.4 Denoting absent arguments

A null pointer or an unallocated allocatable can be used to denote an absent non-allocatable non-pointer optional argument. For example, in

```
interface
  subroutine s(x)
    real, optional :: x
  end subroutine
end interface
⋮
call s(null())
```

the `null()` reference is treated as it if were not present.

This is useful in the slightly contrived situation where one has a procedure with many optional arguments, together with pointers or allocatables to be passed as actual arguments only if associated or allocated. In the absence of this facility one needs a 2^n-way set of nested `if` constructs, where n is the number of local variables in question. Figure A.3 provides an outline of how this process works. In that example, the new feature allows the call to `process_work` to be a single statement; without the feature that call would need to be the unreadably complicated nested `if` constructs shown in Figure A.4.

Figure A.3 Absent optional denotation.

```
subroutine top(x, a, b)
   real :: x
   real, optional, target :: a(:), b(:)
   real, allocatable :: worka(:), workb1(:), workb2(:)
   real, pointer :: pivotptr
   ⋮ (Code to conditionally allocate worka etc. elided.)
   call process_work(x, worka, workb1, workb2, pivot)
end subroutine
subroutine process_work(x, wa, wb1, wb2, pivot)
   real :: x
   real, optional :: wa(:), wb1(:) , wb2(:), pivot
```

It is true that in this example making the dummy variables in `process_work` variously `allocatable` or `pointer` would achieve the same ends, but other callers of `process_work`

Figure A.4 Huge unreadable nested `if` constructs.

```
if (allocated(worka)) then
  if (allocated(workb1)) then
    if (allocated(workb2)) then
      if (associated(pivot)) then
        call process_work(x, worka, workb1, workb2, pivot)
      else
        call process_work(x, worka, workb1, workb2)
      end if
    else if (associated(pivot)) then
      call process_work(x, worka, workb1, pivotptr=pivot)
    else
      call process_work(x, worka, workb1)
    end if
  ⋮ (Remainder of huge nested if construct elided.)
```

might have different mixtures of `allocatable` and `pointer`, or indeed wish to pass plain variables.

The intrinsic functions `count`, `lbound`, `lcobound`, `ubound`, and `ucobound` have an optional argument `dim` whose presence affects the rank of the result. Therefore, the corresponding actual argument is not permitted to be an optional dummy argument name. For the same reason, it is not permitted to be a null pointer or an unallocated allocatable (that is, the feature described in this section is not operative).

A.9.5 Alternative form of complex constant

A complex constant may be written with a named constant of type real or integer for its real part, imaginary part, or both. For example,

```
real, parameter    :: zero = 0, one = 1
complex, parameter :: i = (zero, one)
```

However, no sign is allowed with a name, so although `(0,-1)` is a perfectly good complex constant, `(zero,-one)` is invalid.

Since the intrinsic function `cmplx` is permitted to appear in a constant expression there is very little use for this feature.

B. Obsolescent and deleted features

B.1 Features obsolescent in Fortran 95

The features of this section are described by the Fortran 95 standard to be obsolescent. Their replacements are described in the relevant subsections.

B.1.1 Fixed source form

In the old fixed source form, each statement consists of one or more **lines** exactly 72 characters long,[1] and each line is divided into three **fields**. The first field consists of positions 1 to 5 and may contain a **statement label**. A Fortran statement may be written in the third fields of up to 256 consecutive lines (only 20 in versions prior to Fortran 2003). The first line of a multi-line statement is known as the **initial line** and the succeeding lines as **continuation lines**.

A non-comment line is an initial line or a continuation line depending on whether there is a character other than zero or blank in position 6 of the line, which is the second field. The first field of a continuation line must be blank. The ampersand is not used for continuation.

The third field, from positions 7 to 72, is reserved for the Fortran statements themselves. Note that if a construct is named, the name must be placed here and not in the label field.

Except in a character context, blanks are insignificant.

The presence of an asterisk (`*`) or a character `c` in position 1 of a line indicates that the whole line is commentary. An exclamation mark indicates the start of commentary, except in position 6, where it indicates continuation.

Several statements separated by a semicolon (`;`) may appear on one line. The semicolon may not, in this case, be in column 6, where it would indicate continuation. Only the first of the statements on a line may be labelled. A semicolon that is the last non-blank character of a line, or the last non-blank character ahead of commentary, is ignored.

A program unit `end` statement must not be continued, and any other statement with an initial line that appears to be a program unit `end` statement must not be continued.

A processor may restrict the appearance of its defined control characters, if any, in this source form.

In applications where a high degree of compatibility between the old and the new source forms is required, observance of the following rules can be of great help:

[1] This limit is processor dependent if the line contains characters other than those of the default type.

- confine statement labels to positions 1 to 5 and statements to positions 7 to 72;
- treat blanks as being significant;
- use only `!` to indicate a comment (but not in position 6);
- for continued statements, place an ampersand in both position 73 of a continued line and position 6 of a continuing line.

The fixed source form has been replaced by the free source form (Section 2.4).

B.1.2 Computed go to

A form of branch statement is the computed `go to`, which enables one path among many to be selected, depending on the value of a scalar integer expression. The general form is

`go to (`sl_1`, `sl_2`, `sl_3`, ...) [ , ]` *intexpr*

where sl_1, sl_2, sl_3, etc. are labels of statements in the same scoping unit, and *intexpr* is any scalar integer expression. The same statement label may appear more than once. An example is the statement

```
go to (6,10,20) i(k)**2+j
```

which references three statement labels. When the statement is executed, if the value of the integer expression is 1, the first branch will be taken, and control is transferred to the statement labelled 6. If the value is 2, the second branch will be taken, and so on. If the value is less than 1, or greater than 3, no branch will be taken, and the next statement following the `go to` will be executed.

This statement is replaced by the `case` construct (Section 4.3).

B.1.3 Character length specification with character*

Alternatives for default characters to

`character (` *[* `len=` *]len-value*`)`

as a *type* in a type declaration, `function`, `implicit`, or component definition statement are

`character*(`*len-value*`) [ , ]`

and

`character*`*len [,]*

where *len* is an integer literal constant without a specified kind value, and the optional comma is permitted only in a type declaration statement and only when `::` is absent:

```
character*20, word, letter*1
```

B.1.4 Data statements among executables

The `data` statement may be placed among the executable statements, but such placement is rarely used and not recommended, since data initialization properly belongs with the specification statements.

B.1.5 Statement functions

It may be that within a single program unit there are repeated occurrences of a computation which can be represented as a single statement. For instance, to calculate the parabolic function represented by

$$y = a + bx + cx^2$$

for different values of x, but with the same coefficients, there may be references to

```
y1 = 1. + x1*(2. + 3.*x1)
⋮
y2 = 1. + x2*(2. + 3.*x2)
⋮
```

etc. In Fortran 77 it was more convenient to invoke a so-called **statement function** (now better coded as an internal function, Section 5.6), which must appear after any `implicit` and other relevant specification statements and before the executable statements. The example above would become

```
parab(x) = 1. + x*(2. + 3.*x)
⋮
y1 = parab(x1)
⋮
y2 = parab(x2)
```

Here, `x` is a dummy argument, which is used in the definition of the statement function. The variables `x1` and `x2` are actual arguments to the function.

The general form is

`function-name (` *[dummy-argument-list]* `)` `=` *scalar-expr*

where the *function-name* and each *dummy-argument* must be specified, explicitly or implicitly, to be scalar data objects. The function must not be of a parameterized derived type (Section 13.2). To make it clear that this is a statement function and not an assignment to a host array element, we recommend declaring the type by placing the *function-name* in a type declaration statement; this is *required* whenever a host entity has the same name.

The *scalar-expr* must be composed of constants, references to scalar variables, references to functions, and intrinsic operations. If there is a reference to a function, the function must not be a transformational intrinsic nor require an explicit interface, the result must be scalar, and any array argument must be a named array. A reference to a non-intrinsic function must not require an explicit interface. A named constant that is referenced or an array of which an element is referenced must be declared earlier in the scoping unit or be accessed by use or host association. A scalar variable referenced may be a dummy argument of the statement function or a variable that is accessible in the scoping unit. A dummy argument of the host procedure must not be referenced unless it is a dummy argument of the main entry or of an `entry` that precedes the statement function. If any entity is implicitly typed, a subsequent type declaration must confirm the type and type parameters. The dummy arguments are scalar and have a scope of the statement function statement only.

A statement function always has an implicit interface and may not be supplied as a procedure argument. It may appear within an internal procedure, and may reference other statement functions appearing before it in the same scoping unit, but not itself nor any appearing after. A function reference in the expression must not redefine a dummy argument. A statement function is pure (Section 7.8) if it references only pure functions.

A statement function statement is not permitted in an interface block.

Note that statement functions are irregular in that use and host association are not available.

B.1.6 Assumed character length of function results

A non-recursive external function whose result is scalar, character, and non-pointer may have assumed character length, as in Figure B.1. Such a function is not permitted to specify a defined operation. In a scoping unit that invokes such a function, the interface must be implicit and there must be a declaration of the length, as in Figure B.2, or such a declaration must be accessible by use or host association.

Figure B.1 A function whose result is of assumed character length.

```
function copy(word)
   character(len=*) copy, word
   copy = word
end function copy
```

Figure B.2 Calling a function whose result is of assumed character length.

```
program main
   external copy             ! Interface block not allowed.
   character(len=10) copy
   write (*, *) copy('This message will be truncated')
end program main
```

This facility is included only for compatibility with Fortran 77 and is completely at variance with the philosophy of modern Fortran that the attributes of a function result depend only on the actual arguments of the invocation and on any data accessible by the function through host or use association.

This facility may be replaced by use of a subroutine whose arguments correspond to the function result and the function arguments.

B.1.7 Alternate return

When calling certain types of subroutines it is possible that specific exceptiona conditions will arise that should cause a break in the normal control flow. It is possible to anticipate such conditions, and to code different flow paths following a subroutine call depending on whether the called subroutine has terminated normally or has detected an exceptional or abnormal

condition. This is achieved using the alternate `return` facility which uses the argument list in the following manner. Let us suppose that a subroutine `deal` receives in an argument list the number of cards in a shuffled deck, the number of players, and the number of cards to be dealt to each hand. In the interests of generality, it would be a reasonable precaution for the first executable statement of `deal` to be a check that there is at least one player and that there are, in fact, enough cards to satisfy each player's requirement. If there are no players or insufficient cards, it can signal this to the main program which should then take the appropriate action. This may be written in outline as

```
      call deal(nshuff, nplay, nhand, cards, *2, *3)
      call play
      ⋮
2     ...        ! Handle no-player case
      ⋮
3     ...        ! Handle insufficient-cards case
      ⋮
```

If the cards can be dealt, normal control is returned, and the call to `play` executed. If an exception occurs, control is passed to the statement labelled 2 or 3, at which point some action must be taken – to stop the game or shuffle more cards. The relevant statement label is defined by placing the statement label preceded by an asterisk as an actual argument in the argument list. It must be a label of an executable statement of the same scoping unit. Any number of such alternate returns may be specified, and they may appear in any position in the argument list. Since, however, they are normally used to handle exceptions, they are best placed at the end of the list.

In the called subroutine, the corresponding dummy arguments are asterisks and the alternate `return` is taken by executing a statement of the form

`return` *int-expr*

where *int-expr* is any scalar integer expression. The value of this expression at execution time defines an index to the alternate `return` to be taken, according to its position in the argument list. If *int-expr* evaluates to 2, the second alternate `return` will be taken. If *int-expr* evaluates to a value which is less than 1, or greater than the number of alternate `return`s in the argument list, a normal `return` will be taken. Thus, in `deal`, we may write simply

```
subroutine deal(nshuff, nplay, nhand, cards, *, *)
   ⋮
   if (nplay.le.0) return 1
   if (nshuff .lt. nplay*nhand) return 2
```

This feature is also available for subroutines defined by `entry` statements. It is not available for functions or elemental subroutines.

This feature is replaced by use of an integer argument holding a return code used in a following `case` construct.

B.2 Feature obsolescent in Fortran 2008: Entry statement

A subprogram usually defines a single procedure, and the first statement to be executed is the first executable statement after the header statement. In some cases it is useful to be able to define several procedures in one subprogram, particularly when wishing to share access to some saved local variables or to a section of code. This is possible for external and module subprograms (but not for internal subprograms) by means of the `entry` statement. This is a statement that has the form

`entry` *entry-name [* `(` *[dummy-argument-list]* `)` *[* `result(`*result-name*`)` *]]*

and may appear anywhere between the header line and `contains` (or `end` if it has no `contains`) statement of a subprogram, except within a construct. The `entry` statement provides a procedure with an associated dummy argument list, exactly as does the `subroutine` or `function` statement, and these arguments may be different from those given on the `subroutine` or `function` statement. Execution commences with the first executable statement following the `entry` statement.

In the case of a function, each `entry` defines another function, whose characteristics (that is, shape, type, type parameters, and whether a pointer) are given by specifications for the *result-name* (or *entry-name* if there is no `result` clause). If the characteristics are the same as for the main entry, a single variable is used for both results; otherwise, they must not be allocatable, must not be pointers, must be scalar, and must both be one of the default integer, default real, double precision real (Appendix A.6), or default complex types, and they are treated as equivalenced. The `result` clause plays exactly the same role as for the main entry.

Each entry is regarded as defining another procedure, with its own name. The names of all these procedures and their result variables (if any) must be distinct. The name of an entry has the same scope as the name of the subprogram. It must not be the name of a dummy argument of any of the procedures defined by the subprogram. An `entry` statement is not permitted in an interface block; there must be another body for each entry whose interface is wanted, using a `subroutine` or `function` statement, rather than an `entry` statement.

An `entry` is called in exactly the same manner as a subroutine or function, depending on whether it appears in a subroutine subprogram or a function subprogram. An example is given in Figure B.3, which shows a search function with two entry points. We note that `looku` and `looks` are synonymous within the function, so that it is immaterial which value is set before the return.

None of the procedures defined by a subprogram is permitted to reference itself, unless the keyword `recursive` is present on the `subroutine` or `function` statement. For a function, such a reference must be indirect unless there is a `result` clause on the `function` or `entry` statement. If a procedure may be referenced directly in the subprogram that defines it, the interface is explicit in the subprogram.

The name of an `entry` dummy argument that appears in an executable statement preceding the `entry` statement in the subprogram must also appear in a `function`, `subroutine`, or `entry` statement that precedes the executable statement. Also, if a dummy argument is used to define the array size or character length of an object, the object must not be referenced unless the argument is present in the procedure reference that is active.

During the execution of one of the procedures defined by a subprogram, a reference to a dummy argument is permitted only if it is a dummy argument of the procedure referenced.

Figure B.3 A search function with two entry points.

```
      function looku(list, member)
      integer looku, list(:), member, looks
!
!     To locate member in an array list.
!     If list is unsorted, entry looku is used;
!     if list is sorted, entry looks is used.
!
!     List is unsorted.
      do looku = 1, size(list)
         if (list(looku) == member) return
      end do
!
!     Not found.
      looku = 0
      return
!
!     Entry point for sorted list.
!
      entry looks(list, member)
      do looks = 1, size(list)
         if (list(looks) == member) return
         if (list(looks) > member) exit
      end do
!
!     Not found.
      looks = 0
      end function
```

The `entry` statement is made unnecessary by the use of modules (Section 5.5), with each procedure defined by an entry becoming a module procedure.

B.3 Features obsolescent in Fortran 2018

The features of this section are described by the Fortran 2018 standard to be obsolescent. Their replacements are described in the relevant subsections.

B.3.1 The forall statement and construct

When elements of an array are assigned values by a `do` construct such as

```
do i = 1, n
   a(i, i) = 2.0 * x(i)      ! a is rank-2 and x rank-1
end do
```

the processor is required to perform each successive iteration in order and one after the other. This represents a potentially severe impediment to optimization on a parallel processor so, for this purpose, Fortran has the `forall` statement. The above loop can be written as

```
forall (i = 1:n) a(i, i) = 2.0 * x(i)
```

which specifies that the set of expressions denoted by the right-hand side of the assignment is first evaluated in any order, and the results are then assigned to their corresponding array elements, again in any order of execution. The `forall` statement may be considered to be an array assignment expressed with the help of indices. In this particular example, we note also that this operation could not otherwise be represented as a simple array assignment. Other examples of the `forall` statement are

```
forall (i = 1:n, j = 1:m)                 a(i, j) = i + j
forall (i = 1:n, j = 1:n, y(i, j) /= 0.) x(j, i) = 1.0/y(i, j)
```

where, in the second statement, we note the masking condition – the assignment is not carried out for zero elements of `y`.

The `forall` construct also exists. The `forall` equivalent of the array assignments

```
a(2:n-1, 2:n-1) = a(2:n-1, 1:n-2) + a(2:n-1, 3:n)   &
                + a(1:n-2, 2:n-1) + a(3:n, 2:n-1)
b(2:n-1, 2:n-1) = a(2:n-1, 2:n-1)
```

is

```
forall (i = 2:n-1, j = 2:n-1)
   a(i, j) = a(i, j-1) + a(i, j+1) + a(i-1, j) + a(i+1, j)
   b(i, j) = a(i, j)
end forall
```

This sets each internal element of `a` equal to the sum of its four nearest neighbours and copies the result to `b`. The `forall` version is more readable. Note that each assignment in a `forall` is like an array assignment; the effect is as if all the expressions were evaluated in any order, held in temporary storage, then all the assignments performed in any order. Each statement in a `forall` construct must fully complete before the next can begin.

A `forall` statement or construct may contain pointer assignments. An example is

```
type element
   character(32), pointer :: name
end type element
type(element)          :: chart(200)
character(32), target :: names(200)
⋮
forall (i = 1:200)
   chart(i)%name => names(i)
end forall
```

Note that there is no array syntax for performing, as in this example, an array of pointer assignments.

As with all constructs, `forall` constructs may be nested. The sequence

```
forall (i = 1:n-1)
   forall (j = i+1:n)
      a(i, j) = a(j, i)          ! a is a rank-2 array
   end forall
end forall
```

assigns the transpose of the lower triangle of `a` to the upper triangle of `a`.

A `forall` construct can include a `where` statement or construct. Each statement of a `where` construct is executed in sequence. An example with a `where` statement is

```
forall (i = 1:n)
   where ( a(i, :) == 0) a(i, :) = i
   b(i, :) = i / a(i, :)
end forall
```

Here, each zero element of `a` is replaced by the value of the row index and, following this complete operation, the elements of the rows of `b` are assigned the reciprocals of the corresponding elements of `a` multiplied by the corresponding row index.

The complete syntax of the `forall` construct is

[name `:` *]* `forall (` *[type-spec* `::` *]* *index-spec-list [* `,` *scalar-logical-expr]* `)`
[body]
`end forall` *[name]*

where *index-spec* is

index-variable-name `=` *lower* `:` *upper [* `:` *stride]*

Each *index-variable-name* is a scalar variable local to the loop, so has no effect on any variable with the same name that might exist outside the loop; however, if *type-spec* is omitted, it has the type and kind it would have if it were such a variable. An index variable must not be redefined within the construct. Within a nested construct, each index variable must have a distinct name. The expressions *lower*, *upper*, and *stride* (*stride* is optional but must be nonzero when present) are scalar integer expressions and form a sequence of values as for a section subscript (Section 7.12); they may not reference any index variable of the same statement but may reference an index variable of an outer `forall`. Once these expressions have been evaluated, the *scalar-logical-expr*, if present, is evaluated for each combination of index values. Those for which it has the value `.true.` are active in each statement of the construct. The *name* is the optional construct name; if present, it must appear on both the `forall` and the `end forall` statements.

The *body* itself consists of one or more assignment statements, pointer assignment statements, `where` statements or constructs, and further `forall` statements or constructs. The subobject on the left-hand side of each assignment in the *body* should reference each index variable of the constructs it is contained in as part of the identification of the subobject, whether it be a non-pointer variable or a pointer object.[2]

[2]This is not actually a requirement, but any missing index variable would need to be restricted to a single value to satisfy the requirements of the final paragraph of this section. For example, the statement

```
forall (i = i1:i2, j = j1:j2) a(j) = a(j) + b(i, j)
```

is valid only if `i1` and `i2` have the same value.

In the case of a defined assignment statement, the subroutine that is invoked must not reference any variable that becomes defined by the statement, nor any pointer object that becomes associated.

A `forall` construct whose body is a single assignment or pointer assignment statement may be written as a single `forall` statement.

Procedures may be referenced within the scope of a `forall`, both in the logical scalar expression that forms the optional mask or, directly or indirectly (for instance as a defined operation or assignment), in the body of the construct. However, since there is a possibility that such a reference might have side-effects, presenting a severe impediment to optimization on a parallel processor (as the order of execution of the assignments could affect the results), all such procedures must be pure (see Section 7.8).

As in assignments to array sections (Section 7.12), it is not allowed to make a many-to-one assignment. The construct

```
forall (i = 1:10)
   a(index(i)) = b(i)       ! a, b and index are arrays
end forall
```

is valid if and only if `index(1:10)` contains no repeated values. Similarly, it is not permitted to associate more than one target with the same pointer.

As we have seen, in a `forall` statement or construct all the index variables are local to the construct; for example, in

```
idx = 3
forall (idx=100:200) a(idx, idx) = idx**2
print *, idx
```

the value `3` is printed because the `idx` within the `forall` is not the same as the `idx` outside the `forall`. However, the `idx` within the `forall` has the same type and kind as the one outside would have if it existed; and if it does exist, it has to be a scalar integer variable. This is a bit inconvenient, and although there is no other connection between the index variable and any entity outside the `forall`, changing the type or kind of the outer entity would affect the index variable too.

For this reason we recommend specifying the type and kind of the index variables on the `forall` statement itself; for example, in

```
complex :: i(100)
⋮
forall (integer(int64) :: i=1:2_int64**32) a(i) = i*2.0**(-32)
```

the outer variable `i` is a complex array, but has no effect on the index variable because of the declaration in the `forall` statement.

The `forall` construct and statement were added to the language in the hope of efficient execution, but this has not happened, and they have become redundant with the introduction of the `do concurrent` construct and the use of pointer rank remapping.

B.3.2 The equivalence statement

The `equivalence` statement specifies that a given storage area may be shared by two or more objects. For instance,

```
real aa, angle, alpha, a(3)
equivalence (aa, angle), (alpha, a(1))
```

allows `aa` and `angle` to be used interchangeably in the program text, as both names now refer to the same storage location. Similarly, `alpha` and `a(1)` may be used interchangeably.

It is possible to equivalence arrays together. In

```
real a(3,3), b(3,3), col1(3), col2(3), col3(3)
equivalence (col1, a, b), (col2, a(1,2)), (col3, a(1,3))
```

the two arrays `a` and `b` are equivalenced, and the columns of `a` (and hence of `b`) are equivalenced to the arrays `col1`, etc. We note in this example that more than two entities may be equivalenced together, even in a single declaration.

It is possible to equivalence variables of the same intrinsic type and kind type parameter or of the same derived type having the `sequence` attribute. It is also possible to equivalence variables of different types if both have numeric storage association or both have character storage association (see Appendix A.2). Default character variables need not have the same length, as in

```
character(len=4) a
character(len=3) b(2)
equivalence (a, b(1)(3:))
```

where the character variable `a` is equivalenced to the last four characters of the six characters of the character array `b`. Zero character length is not permitted. An example for different types is

```
integer i(100)
real x(100)
equivalence (i, x)
```

where the arrays `i` and `x` are equivalenced. This might be used, for instance, to save storage space if `i` is used in one part of a program unit and `x` separately in another part. This is a highly dangerous practice, as considerable confusion can arise when one storage area contains variables of two or more data types, and program changes may be made very difficult if the two uses of the one area are to be kept distinct.

Types with default initialization are permitted, provided each initialized component has the same type, type parameters, and value in any pair of equivalenced objects. Coarrays are not permitted.

All the various combinations of types that may be equivalenced have been described. No other is allowed. Also, apart from double precision real and the default numeric types, equivalencing objects that have different kind type parameters is not allowed. The general form of the statement is

`equivalence` (*object*, *object-list*) *[* , (*object*, *object-list*) *]* ...

where each *object* is a variable name, array element, or substring. An object must be a variable and must not be a dummy argument, a function result, a pointer, an object with a pointer component at any level of component selection, an allocatable object, an automatic object, a function, a structure component, a structure with an ultimate allocatable component, or a subobject of such an object. Each array subscript and character substring range must

be a constant expression. The interpretation of an array name is identical to that of its first element. An equivalence object must not have the `target` attribute.

The objects in an equivalence set are said to be **storage associated**. Those of nonzero length share the same first storage unit. Those of zero length are associated with each other and with the first storage unit of those of nonzero length. An `equivalence` statement may cause other parts of the objects to be associated, but not such that different subobjects of the same object share storage. For example,

```
real a(2), b
equivalence (a(1), b), (a(2), b)  ! Prohibited
```

is not permitted. Also, objects declared in different scoping units must not be equivalenced. For example,

```
use my_module, only : xx
real bb
equivalence(xx, bb)               ! Prohibited
```

is not permitted.

The various uses to which `equivalence` was put are replaced by automatic arrays, allocatable arrays, pointers (reuse of storage, Chapter 6 and Section 7.3), pointers as aliases (storage mapping, Section 7.14), and the `transfer` function (mapping of one data type onto another, Section 9.11).

B.3.3 The common block

We have seen in Chapter 5 how two program units are able to communicate by passing variables or values of expressions between them via argument lists or by using modules. It is also possible to define areas of storage known as `common` blocks. Each has a storage sequence and may be either named or unnamed, as shown by the simplified syntax of the `common` specification statement:

`common` *[/ [cname] /] vlist*

in which *cname* is an optional name and *vlist* is a list of variable names, each optionally followed by an array bounds specification. An unnamed `common` block is known as a **blank** `common` block. Examples of each are

```
common /hands/ nshuff, nplay, nhand, cards(52)
```

and

```
common // buffer(10000)
```

in which the named `common` block `hands` defines a data area containing the quantities which might be required by the subroutines of a card-playing program, and the blank `common` defines a large data area which might be used by different subroutines as a buffer area.

The name of a `common` block has global scope and must differ from that of any other global entity (external procedure, program unit, or `common` block). It may, however, be the same as that of a local entity other than a named constant or intrinsic procedure.

No object in a `common` block may have the `parameter` attribute or be a dummy argument, an automatic object, an allocatable object, a structure with an ultimate allocatable component,

a coarray, a polymorphic pointer, or a function. An array may have its bounds declared either in the `common` statement or in a type declaration or `dimension` statement. If it is a non-pointer array, the bounds must be declared explicitly and with constant expressions. If it is an array pointer, however, the bounds may not be declared in the `common` statement itself. If an object is of derived type, the type must have the `sequence` or `bind` attribute and must not have default initialization.

In order for a subroutine to access the variables in the data area, it is sufficient to insert the `common` definition in each scoping unit which requires access to one or more of the entities in the list. In this fashion, the variables `nshuff`, `nplay`, `nhand`, and `cards` are made available to those scoping units. No variable may appear more than once in all the `common` blocks in a scoping unit.

Usually a `common` block contains identical variable names in all its appearances, but this is not necessary. In fact, the shared data area may be partitioned in quite different ways in different subroutines, using different variable names. They are said to be storage associated. It is thus possible for one subroutine to contain a declaration

```
common /coords/ x, y, z, i(10)
```

and another to contain a declaration

```
common /coords/ i, j, a(11)
```

This means that a reference to `i(1)` in the first subroutine is equivalent to a reference to `a(2)` in the second. Through multiple references via use or host association, this can even happen in a single subroutine. This manner of coding is both untidy and dangerous, and every effort should be made to ensure that all declarations of a given `common` block declaration are identical in every respect. In particular, the presence or absence of the `target` attribute is required to be consistent, since otherwise a compiler would have to assume that everything in `common` has the `target` attribute, in case it has it in another program unit.

A further practice that is permitted but which we do not recommend is to mix different storage units in the same `common` block. When this is done, each position in the storage sequence must always be occupied by a storage unit of the same category.

The total number of storage units must be the same in each occurrence of a named `common` block, but blank `common` is allowed to vary in size and the longest definition will apply for the complete program.

Yet another practice to be avoided is to use the full syntax of the `common` statement:

`common` *[/ [cname] / vlist [[,] / [cname] / vlist] ...*

which allows several `common` blocks to be defined in one statement, and a single `common` block to be declared in parts. A combined example is

```
common /pts/x,y,z /matrix/a(10,10),b(5,5) /pts/i,j,k
```

which is equivalent to

```
common /pts/ x, y, z, i, j, k
common /matrix/ a(10,10), b(5,5)
```

which is certainly a more understandable declaration of two shared data areas. The only need for the piecewise declaration of one block was when the former limit of 39 continuation lines was otherwise too low.

The common statement may be combined with the equivalence statement, as in the example

```
real a(10), b
equivalence (a,b)
common /change/ b
```

In this case, a is regarded as part of the common block, and its length is extended appropriately. Such an equivalence must not cause data in two different common blocks to become storage associated, it must not cause an extension of the common block except at its tail, and two different objects or subobjects in the same common block must not become storage associated. It must not cause an object to become associated with an object in a common block if it has a property that would prevent it being an object in a common block.

A common block may be declared in a module, and its variables accessed by use association. Variable names in a common block in a module may be declared to have the private attribute, but this does not prevent associated variables being declared elsewhere through other common statements.

An individual variable in a common block may not be given the save attribute, but the whole block may. If a common block has the save attribute in any scoping unit other than the main program, it must have the save attribute in all such scoping units. The general form of the save statement is

save *[[::] saved-entity-list]*

where *saved-entity* is *variable-name* or *common-block-name*. A simple example is

```
save /change/
```

A blank common always has the save attribute.

Data in a common block without the save attribute become undefined on return from a subprogram unless the block is also declared in the main program or in another subprogram that is in execution.

The use of modules (Section 5.5) obviates the need for common blocks.

B.3.4 The block data program unit

Non-pointer variables in named common blocks may be initialized in data statements, but such statements must be collected into a special type of program unit, known as a block data program unit. It must have the form

block data *[block-data-name]*
    [specification] ...
end *[* block data *[block-data-name]]*

where each *specification* is a derived-type definition or an asynchronous, bind, common, data, dimension, equivalence, implicit, intrinsic, parameter, pointer, save, target, type declaration (including double precision), use, or volatile statement. A type declaration statement must not specify the allocatable, bind, external, intent, optional, private, or public attributes. An example is

```
block data
   common /axes/ i,j,k
   data i,j,k /1,2,3/
end block data
```

in which the variables in the `common` block `axes` are defined for use in any other scoping unit which accesses them.

It is possible to collect many `common` blocks and their corresponding `data` statements together in one `block data` program unit. However, it may be a better practice to have several different `block data` program units, each containing `common` blocks which have some logical association with one another. To allow for this, `block data` program units may be named in order to be able to distinguish them. A complete program may contain any number of `block data` program units, but only one of them may be unnamed. A `common` block must not appear in more than one `block data` program unit. It is not possible to initialize blank `common`.

The name of a `block data` program unit may appear in an `external` statement. When a processor is loading program units from a library, it may need such a statement in order to load the `block data` program unit.

The use of modules (Section 5.5) obviates the need for `block data`.

B.3.5 The labelled do construct

A further form of the `do` construct (Section 4.4) makes use of a statement label to identify the end of the construct. In this case, the terminating statement may be either a labelled `end do` statement or a labelled `continue` ('do nothing') statement.[3] The label is, in each case, the same as that on the `do` statement itself. The label on the `do` statement may be followed by a comma. Simple examples are

```
      do 10 i = 1, n
         ⋮
10 end do
```

and

```
      do 20 i = 1, j
         do 10, k = 1, l
            ⋮
10       continue
20 continue
```

As shown in the second example, each loop must have a separate label.

Labelled `do` statements are now redundant with the use of a name for the construct and the use of the `cycle` statement.

[3]The `continue` statement is not limited to being the last statement of a `do` construct; it may appear anywhere among the executable statements.

B.3.6 Specific names of intrinsic procedures

While all of the intrinsic procedures are generic, some of the intrinsic functions also have **specific names** for specific versions, which are listed in Tables B.1 and B.2. In the tables, 'Character' stands for default character, 'Integer' stands for default integer, 'Real' stands for default real, 'Double' stands for double precision real, and 'Complex' stands for default complex. Those functions in Table B.2 may be passed as actual arguments to a procedure, or be associated with a procedure pointer, provided they are specified in an `intrinsic` statement (Section 9.1.3).

Table B.1. Specific intrinsic functions not usable as arguments or targets.

Description	Generic form	Specific name	Argument type	Function type
Conversion to integer	`int(a)`	`int`	Real	Integer
		`ifix`	Real	Integer
		`idint`	Double	Integer
Conversion to real	`real(a)`	`real`	Integer	Real
		`float`	Integer	Real
		`sngl`	Double	Real
max(`a1,a2,...`)	`max(a1,a2,...)`	`max0`	Integer	Integer
		`amax1`	Real	Real
		`dmax1`	Double	Double
	`int(max(a1,a2,...))`	`amax0`	Integer	Real
	`real(max(a1,a2,...))`	`max1`	Real	Integer
min(`a1,a2,...`)	`min(a1,a2,...)`	`min0`	Integer	Integer
		`amin1`	Real	Real
		`dmin1`	Double	Double
	`int(min(a1,a2,...))`	`amin0`	Integer	Real
	`real(min(a1,a2,...))`	`min1`	Real	Integer
lge(`string_a,string_b`)	`lge(string_a,string_b)`	`lge`	Character	Logical
lgt(`string_a,string_b`)	`lgt(string_a,string_b)`	`lgt`	Character	Logical
lle(`string_a,string_b`)	`lle(string_a,string_b)`	`lle`	Character	Logical
llt(`string_a,string_b`)	`llt(string_a,string_b)`	`llt`	Character	Logical

Table B.2. Specific intrinsic functions usable as arguments or targets.

Description	Generic form	Specific name	Argument type	Function type
Absolute value of `a` times sign of `b`	`sign(a,b)`	`isign`	Integer	Integer
		`sign`	Real	Real
		`dsign`	Double	Double
max(`x-y`,0)	`dim(x,y)`	`idim`	Integer	Integer
		`dim`	Real	Real
		`ddim`	Double	Double
`x` × `y`		`dprod(x,y)`	Real	Double
Truncation	`aint(a)`	`aint`	Real	Real
		`dint`	Double	Double
Nearest whole number	`anint(a)`	`anint`	Real	Real
		`dnint`	Double	Double
Nearest integer	`nint(a)`	`nint`	Real	Integer
		`idnint`	Double	Integer
Absolute value	`abs(a)`	`iabs`	Integer	Integer
		`abs`	Real	Real
		`dabs`	Double	Double
		`cabs`	Complex	Real
Remainder modulo `p`	`mod(a,p)`	`mod`	Integer	Integer
		`amod`	Real	Real
		`dmod`	Double	Double
Square root	`sqrt(x)`	`sqrt`	Real	Real
		`dsqrt`	Double	Double
		`csqrt`	Complex	Complex
Exponential	`exp(x)`	`exp`	Real	Real
		`dexp`	Double	Double
		`cexp`	Complex	Complex
Natural logarithm	`log(x)`	`alog`	Real	Real
		`dlog`	Double	Double
		`clog`	Complex	Complex
Common logarithm	`log10(x)`	`alog10`	Real	Real
		`dlog10`	Double	Double
Sine	`sin(x)`	`sin`	Real	Real
		`dsin`	Double	Double
		`csin`	Complex	Complex

Table B.2 (cont.). Specific intrinsic functions usable as arguments or targets.

Description	Generic form	Specific name	Argument type	Function type
Cosine	`cos(x)`	`cos`	Real	Real
		`dcos`	Double	Double
		`ccos`	Complex	Complex
Tangent	`tan(x)`	`tan`	Real	Real
		`dtan`	Double	Double
Arcsine	`asin(x)`	`asin`	Real	Real
		`dasin`	Double	Double
Arccosine	`acos(x)`	`acos`	Real	Real
		`dacos`	Double	Double
Arctangent	`atan(x)`	`atan`	Real	Real
		`datan`	Double	Double
	`atan2(y,x)`	`atan2`	Real	Real
		`datan2`	Double	Double
Hyperbolic sine	`sinh(x)`	`sinh`	Real	Real
		`dsinh`	Double	Double
Hyperbolic cosine	`cosh(x)`	`cosh`	Real	Real
		`dcosh`	Double	Double
Hyperbolic tangent	`tanh(x)`	`tanh`	Real	Real
		`dtanh`	Double	Double
Imaginary part	`aimag(z)`	`aimag`	Complex	Real
Complex conjugate	`conjg(z)`	`conjg`	Complex	Complex
Character length	`len(s)`	`len`	Character	Integer
Starting position	`index(s,t)`	`index`	Character	Integer

Specific names for intrinsic functions have been redundant since Fortran 77. They are now obsolescent as they all have generic names.

B.4 Features deleted in Fortran 95

The features listed in this section were deleted from the Fortran 95 language entirely. They are fully described in early editions of this book.

Non-integer do indices The `do` variable and the expressions that specify the limits and stride of a `do` construct or an implied-`do` in an I/O statement could be of type default real or double precision real.

Assigned go to and assigned formats Another form of branch statement was actually written in two parts, an `assign` statement and an assigned `go to` statement. One use of

the `assign` statement is replaced by character expressions to define format specifiers (Section 10.4).

Branching to an end if statement It was permissible to branch to an `end if` statement from outside the construct that it terminates. A branch to the following statement is a replacement for this practice.

The pause statement At certain points in the execution of a program it was possible to pause, in order to allow some possible external intervention in the running conditions to be made.

H edit descriptor The `H` (or `h`) edit descriptor provided an early form of the character string edit descriptor.

B.5 Feature deleted in Fortran 2003: Carriage control

Fortran's formatted output statements were originally designed for line printers, with their concept of lines and pages of output. On such a device, the first character of each output record had to be of default kind and was not printed but interpreted as a **carriage control character**. The carriage control characters defined by the standard were:

b	to start a new line
`+`	to remain on the same line (overprint)
`0`	to skip a line
`1`	to advance to the beginning of the next page

The feature is no longer part of the standard and is fully described in early editions of this book.

B.6 Features deleted in Fortran 2018

The features listed in this section were deleted from the Fortran 2018 language entirely. Although it can be expected that compilers will continue to support these features for some period, their use should be completely avoided to ensure long-term portability and to avoid unnecessary compiler warning messages. They are fully described in earlier editions of this book.

Arithmetic if statement The arithmetic `if` provided a three-way branching mechanism, depending on whether an arithmetic expression has a value which is less than, equal to, or greater than zero. It is incompatible with IEEE arithmetic, and thus virtually all modern computers, because a NaN is neither less than, equal to, nor greater than zero. It involved the use of labels, which can hinder optimization and make code hard to read and maintain. It is replaced by the `if` statement and construct (Section 4.2).

Shared do-loop termination A `do`-loop could be terminated on a labelled statement other than an `end do` or `continue`. Nested `do`-loops could also share the same labelled

terminal statement, in which case all the usual rules for nested blocks held, but a branch to the label had to be from within the innermost loop. This offered considerable scope for confusion and unexpected errors. Note that labelled statements are still included as an obsolescent part of the language; they are now limited such that they form properly nested blocks.

C. Object-oriented list example

A recurring problem in computing is the need to manipulate a dynamic data structure. This might be a simple homogeneous linked list like the one encountered in Section 2.12, but often a more complex structure is required.

The example in this appendix consists of a module that provides two types – a list type `anylist` and an item type `anyitem` – for building heterogeneous doubly linked linear lists, plus a simple item constructor function `newitem`. Operations on the list or on items are provided by type-bound procedures. Each list item has a scalar value which may be of any type; when creating a new list item, the required value is copied into the item. A list item can be in at most one list at a time.

List operations include inserting a new item at the beginning or end of the list, returning the item at the beginning or end of the list, and counting, printing, or deleting the whole list.

Operations on an item include removing it from a list, returning the next or previous item on the list, changing the value of the item, and printing or deleting the item. When traversing the list backwards (via the `prev` function), the list is circular; that is, the last item on the list is previous to the first. When traversing the list forwards (via the `next` function), a null pointer is returned after the last item.

Internally, the module uses private pointer components (`firstptr`, `nextptr`, `prevptr`, and `upptr`) to maintain the structure of the lists.

The item `print` operation may be overridden in an extension to `anyitem` to provide printing capability for user-defined types; this is demonstrated by the type `myitem`. All the other procedures are non-overridable, so that extending the list type cannot break the list structure.

The source code is available at `ftp://ftp.numerical.rl.ac.uk/pub/MRandC/oo.f90`

```
module anylist_m
  !
  ! Module for a list type that can contain items with any scalar value.
  ! Values are copied into the list items.
  !
  ! A list item can be in at most one list at a time.
  !
  implicit none
  private
  public :: anylist, anyitem, delete, newitem
  !
  ! type(anylist) is the list header type.
```

```
  !
  type anylist
    class(anyitem), pointer, private :: firstptr => null()
  contains
    procedure, non_overridable :: append
    procedure, non_overridable :: count_list
    procedure, non_overridable :: delete_list
    procedure, non_overridable :: first
    procedure, non_overridable :: last
    procedure, non_overridable :: prepend
    procedure, non_overridable :: print_list
  end type
  !
  ! type(anyitem) is the list item type.
  ! These are allocated by newitem.
  !
  type anyitem
    class(*), allocatable            :: value
    class(anyitem), pointer, private :: nextptr => null(), prevptr => null()
    class(anylist), pointer, private :: upptr => null()
  contains
    procedure, non_overridable :: change
    ! delete has a pointer argument so cannot be type bound
    procedure, non_overridable :: list
    procedure, non_overridable :: next
    procedure, non_overridable :: prev
    procedure                  :: print
    procedure, non_overridable :: remove
  end type

contains
  !
  ! Create a new (orphaned) list item.
  !
  function newitem(something)
    class(*), intent(in)    :: something
    class(anyitem), pointer :: newitem
    allocate (newitem)
    allocate (newitem%value, source=something)
    newitem%prevptr => newitem
  end function
  !
  ! Append an item to a list.
  !
  subroutine append(list, item)
    class(anylist), intent(inout), target :: list
    class(anyitem), target                :: item
    class(anyitem), pointer               :: last
    if (associated(item%upptr)) call remove(item)
```

```
  item%upptr => list
  if (associated(list%firstptr)) then
    last => list%firstptr%prevptr
    last%nextptr => item
    item%prevptr => last
    list%firstptr%prevptr => item
  else
    list%firstptr => item
    item%prevptr => item
  end if
end subroutine
!
! Count how many items there are in a list.
!
integer function count_list(list)
  class(anylist), intent(in) :: list
  class(anyitem), pointer :: p
  count_list = 0
  p => list%firstptr
  do
    if (.not.associated(p)) exit
    count_list = count_list + 1
    p => p%nextptr
  end do
end function
!
! Delete the contents of a list.
!
subroutine delete_list(list)
  class(anylist), intent(inout) :: list
  do
    if (.not.associated(list%firstptr)) exit
    call delete(list%firstptr)
  end do
end subroutine
!
! Return the first element of a list.
!
function first(list)
  class(anylist), intent(in) :: list
  class(anyitem), pointer :: first
  first => list%firstptr
end function
!
! Return the last element of a list
!
function last(list)
  class(anylist), intent(in) :: list
  class(anyitem), pointer :: last
```

```
    last => list%firstptr
    if (associated(last)) last => last%prevptr
  end function
  !
  ! Insert an item at the beginning of a list.
  !
  subroutine prepend(list, item)
    class(anylist), intent(inout), target :: list
    class(anyitem), target                :: item
    if (associated(item%upptr)) call remove(item)
    item%upptr => list
    if (associated(list%firstptr)) then
      item%prevptr => list%firstptr%prevptr
      item%nextptr => list%firstptr
      list%firstptr%prevptr => item
    else
      item%prevptr => item
    end if
    list%firstptr => item
  end subroutine
  !
  ! Print the items in a list.
  !
  subroutine print_list(list, show_item_numbers, show_empty_list)
    class(anylist), intent(in) :: list
    logical, intent(in), optional :: show_item_numbers, show_empty_list
    class(anyitem), pointer :: p
    integer i
    logical :: show_numbers
    if (present(show_item_numbers)) then
      show_numbers = show_item_numbers
    else
      show_numbers = .true.
    end if
    p => list%firstptr
    if (.not.associated(p)) then
      if (present(show_empty_list)) then
        if (show_empty_list) print *, 'List is empty.'
      else
        print *, 'List is empty.'
      end if
    else
      do i=1, huge(i)-1
        if (show_numbers) write (*, 1, advance='no') i
1       format(1x, 'Item ', i0, ':')
        call p%print
        p => p%nextptr
        if (.not.associated(p)) exit
      end do
```

```
  end if
end subroutine
!
! Change the value of an item.
!
subroutine change(item, newvalue)
  class(anyitem), intent(inout) :: item
  class(*), intent(in)          :: newvalue
  deallocate (item%value)
  allocate (item%value, source=newvalue)
end subroutine
!
! Delete an item: removes it from the list and deallocates it.
!
subroutine delete(item)
! We want to deallocate the dummy argument, so it must be a pointer.
! It follows that the procedure cannot be a type-bound procedure.
  class(anyitem), pointer  :: item
  call remove(item)
  deallocate (item)
end subroutine
!
! Return the list that an item is a member of.  Null if an orphan.
!
function list(item)
  class(anyitem), intent(in) :: item
  class(anylist), pointer :: list
  list => item%upptr
end function
!
! Return the next item in the list.
!
function next(item)
  class(anyitem), intent(in) :: item
  class(anyitem), pointer :: next
  next => item%nextptr
end function
!
! Return the previous item in the list,
! or the last item if this one is the first.
!
function prev(item)
  class(anyitem), intent(in) :: item
  class(anyitem), pointer :: prev
  prev => item%prevptr
end function
!
! Print an item.  This is overridable.
!
```

```
  subroutine print(this)
    class(anyitem), intent(in) :: this
    integer length
    select type (v=>this%value)
    type is (character(*))
      length = len(v)
      if (length>40) then
        print 1, length, v(:36)
1       format(1x, 'character(len=', i0, ') = "', a, '"...')
      else
        print *, 'character = "', v, '"'
      end if
    type is (complex)
      print *, 'complex', v
    type is (complex(kind(0d0)))
      print 2, kind(v), v
2     format(1x, 'complex(kind=', i0, ') = (', es23.16, ', ', es23.16, ')')
    type is (real(kind(0d0)))
      print 3, kind(v), v
3     format(1x, 'real(kind=', i0, ') = ', es23.16)
    type is (integer)
      print *, 'integer = ', v
    type is (real)
      print *, 'real = ', v
    type is (logical)
      print *, 'logical = ', v
    class default
      print *, 'unrecognised item type - cannot display value'
    end select
  end subroutine
  !
  ! Remove an item from a list (but keep it and its value).
  !
  subroutine remove(item)
    class(anyitem), intent(inout), target :: item
    class(anylist), pointer :: list
    list => item%upptr
    if (associated(list)) then
      if (associated(item%prevptr, item)) then
        ! Single item in list.
        nullify(list%firstptr)
      else if (.not.associated(item%nextptr)) then
        ! Last item in list.
        list%firstptr%prevptr => item%prevptr
        nullify(item%prevptr%nextptr)
      else if (associated(list%firstptr, item)) then
        ! First item in list.
        list%firstptr => item%nextptr         ! first = next.
        item%nextptr%prevptr => item%prevptr  ! next%prev = last.
```

```
      else
        item%prevptr%nextptr => item%nextptr  ! last%next = item%next.
        item%nextptr%prevptr => item%prevptr  ! next%prev = item%last.
      end if
      item%prevptr => item
    end if
    nullify(item%upptr)
  end subroutine
end module
!
! Module to demonstrate extending anyitem to handle a user-defined type.
!
module myitem_list_m
  use anylist_m
  implicit none
  type, extends(anyitem) :: myitem
  contains
    procedure :: print => myprint
  end type
  type rational
    integer :: numerator = 0
    integer :: denominator = 1
  end type
contains
  !
  ! Version of print that will handle type rational.
  !
  subroutine myprint(this)
    class(myitem), intent(in) :: this
    select type (v=>this%value)
    class is (rational)
      print *, 'rational =', v%numerator, '/', v%denominator
    class default
      call this%anyitem%print
    end select
  end subroutine
  function new_myitem(anything)
    class(*), intent(in) :: anything
    class(myitem), pointer :: new_myitem
    allocate (new_myitem)
    allocate (new_myitem%value, source=anything)
  end function
end module
!
! Demonstration program.
!
program demonstration
  use myitem_list_m
  implicit none
```

```
  type(anylist), target :: list
  class(anyitem), pointer :: p
  !
  ! First demonstrate the most basic workings of a list.
  print *, 'The initial list has', list%count_list(), 'items.'
  call list%append(newitem(17))
  print *, 'The list now has', list%count_list(), 'items.'
  call list%append(newitem('world'))
  print *, 'The list now has', list%count_list(), 'items.'
  call list%prepend(newitem('hello'))
  print *, 'The list now has', list%count_list(), 'items.'
  call list%append(newitem(2.25))
  print *, 'The list now has', list%count_list(), 'items.'
  write (*, '(1x, a)', advance='no') 'The first element is: '
  p => list%first()
  call p%print
  write (*, '(1x, a)', advance='no') 'The last element is: '
  p => list%last()
  call p%print
  print *, 'After deleting the last element, the list contents are:'
  call delete(p)
  call list%print_list
  !
  ! Now delete the old list and make a new one,
  ! with some values from myitem_list_m.
  !
  call list%delete_list
  call list%append(new_myitem('The next value is one third.'))
  call list%append(new_myitem(rational(1,3)))
  call list%append(new_myitem('Where did this number come from?'))
  call list%append(new_myitem(rational(173,13)))
  print *, 'The contents of our new list are:'
  call list%print_list
  !
  ! Now test some of the other procedures, just to prove they work.
  !
  p => list%first()
  p => p%prev()        ! Test prev(), this will be the last item.
  call p%remove        ! Remove the last item.
  call list%prepend(p) ! Put it back, at the beginning of the list.
  p => p%next()        ! Test next(), this will be the second item,
                       ! the one with the string "...third.".
  call p%change((0,1)) ! Replace it with a complex number.
  print *, 'Revised list contents:'
  call list%print_list
  call list%prepend(p) ! Move new item to top
  print *, 'Afer moving item 2 to top, list contents:'
  call list%print_list
end program
```

D. Solutions to exercises

Note: A few exercises have been left to the reader.

Chapter 2

1.

`b is less than m`	true
`8 is less than 2`	false
`* is greater than T`	not determined
`$ is less than /`	not determined
`blank is greater than A`	false
`blank is less than 6`	true

2.

	`x = y`	correct
`3`	`a = b+c ! add`	correct, with commentary
	`word = 'string'`	correct
	`a = 1.0; b = 2.0`	correct
	`a = 15. ! initialize a; b = 22. ! and b`	incorrect (embedded commentary)
	`song = "Life is just&`	correct, initial line
	`& a bowl of cherries"`	correct, continuation
	`chide = 'Waste not,`	incorrect, trailing & missing
	`want not!'`	incorrect, leading & missing
`0`	`c(3:4) = 'up"`	incorrect (invalid statement label; invalid form of character constant)

3.

`-43`	integer	`'word'`	character
`4.39`	real	`1.9-4`	not legal
`0.0001e+20`	real	`'stuff & nonsense'`	character
`4 9`	not legal	`(0.,1.)`	complex
`(1.e3,2)`	complex	`'I can''t'`	character
`'(4.3e9, 6.2)'`	character	`.true._1`	logical[1]
`e5`	not legal	`'shouldn' 't'`	not legal
`1_2`	integer[1]	`"O.K."`	character
`z10`	not legal	`z'10'`	hexadecimal

[1] Legal provided the kind is available.

4.

`name`	legal	`name32`	legal
`quotient`	legal	`123`	not legal
`a182c3`	legal	`no-go`	not legal
`stop!`	not legal	`burn_`	legal
`no_go`	legal	`long__name`	legal

5.

```
real, dimension(11)       ::  a     a(1),a(10),a(11),a(11)
real, dimension(0:11)     ::  b     b(0),b(9),b(10),b(11)
real, dimension(-11:0)    ::  c     c(-11),c(-2),c(-1),c(0)
real, dimension(10,10)    ::  d     d(1,1),d(10,1),d(1,2),d(10,10)
real, dimension(5,9)      ::  e     e(1,1),e(5,2),e(1,3),e(5,9)
real, dimension(5,0:1,4)  ::  f     f(1,0,1),f(5,1,1),f(1,0,2),f(5,1,4)
```

Array constructor: `(/ (i, i = 1,11) /)`

6.

`c(2,3)`	legal	`c(4:3)(2,1)`	not legal
`c(6,2)`	not legal	`c(5,3)(9:9)`	legal
`c(0,3)`	legal	`c(2,1)(4:8)`	legal
`c(4,3)(:)`	legal	`c(3,2)(0:9)`	not legal
`c(5)(2:3)`	not legal	`c(5:6)`	not legal
`c(5,3)(9)`	not legal	`c(,)`	not legal

7.

i)
```
type vehicle_registration
   character(len=3) :: letters
   integer          :: digits
end type vehicle_registration
```

ii)
```
type circle
   real               ::   radius
   real, dimension(2) ::   centre
end type circle
```

iii)
```
type book
   character(len=20)               :: title
   character(len=20), dimension(2) :: author
   integer                         :: no_of_pages
end type book
```

Derived type constants:

```
vehicle_registration('PQR', 123)
circle(15.1, (/ 0., 0. /))
book("Pilgrim's Progress", (/ 'John  ', 'Bunyan' /), 250 )
```

8.

`t`	array	`t(4)%vertex(1)`	scalar
`t(10)`	scalar	`t(5:6)`	array
`t(1)%vertex`	array	`t(5:5)`	array (size 1)

9.

a)
```
integer, parameter   :: twenty = selected_int_kind(20)
integer(kind=twenty) :: counter
```

b)
```
integer, parameter :: high = selected_real_kind(12,100)
real(kind = high)  :: big
```

c)
```
character(kind=2)  :: sign
```

Chapter 3

1.

`a+b`	valid	`-c`	valid
`a+-c`	invalid	`d+(-f)`	valid
`(a+c)**(p+q)`	valid	`(a+c)(p+q)`	invalid
`-(x+y)**i`	valid	`4.((a-d)-(a+4.*x)+1)`	invalid

2.
```
c+(4.*f)
((4.*g)-a)+(d/2.)
a**(e**(c**d))
((a*e)-((c**d)/a))+e
(i .and. j) .or. k
((.not. l) .or. ((.not. i) .and. m)) .neqv. n
((b(3).and.b(1)).or.b(6)).or.(.not.b(2))
```

3.

`3+4/2` = 5	`6/4/2` = 0
`3.*4**2` = 48.	`3.**3/2` = 13.5
`-1.**2` = -1.	`(-1.)**3` = -1.

4.
```
ABCDEFGH
ABCD0123
ABCDEFGu
ABCDbbuu
```
u = unchanged
b = blank

5.

`.not.b(1).and.b(2)`	valid	`.or.b(1)`	invalid
`b(1).or..not.b(4)`	valid	`b(2)(.and.b(3).or.b(4))`	invalid

6.

`d .le.  c`	valid	`p .lt.  t > 0`	invalid
`x-1 /= y`	valid	`x+y < 3 .or.  > 4.`	invalid
`d.lt.c.and.3.0`	invalid	`q.eq.r .and.  s>t`	valid

7.

a) `4*l`

b) `b*h/2.`

c) `4./3.*pi*r**3` (assuming `pi` has value π)

8.

```
integer :: n, one, five, ten, twenty_five
twenty_five = (100-n)/25
ten         = (100-n-25*twenty_five)/10
five        = (100-n-25*twenty_five-10*ten)/5
one         = 100-n-25*twenty_five-10*ten-5*five
```

9.

`a = b + c`	valid
`c = b + 1.0`	valid
`d = b + 1`	invalid
`r = b + c`	valid
`a = r + 2`	valid

10.

`a = b`	valid	`c = a(:,2) + b(5,:5)`	valid
`a = c+1.0`	invalid	`c = a(2,:)  + b(:,5)`	invalid
`a(:,3) = c`	valid	`b(2:,3) = c + b(:5,3)`	invalid

Chapter 4

1.

```
integer                  :: i, j, k, temp
integer, dimension(100) :: reverse
do i = 1,100
   reverse(i) = i
end do
read *, i, j
do k= i, i+(j-i-1)/2
   temp = reverse(k)
   reverse(k) = reverse(j-k+i)
   reverse(j-k+i) = temp
end do
end
```

Note: A simpler method for performing this operation will become apparent in Section 7.12.

2.

```
integer :: limit, f1, f2, f3
read *, limit
f1 = 1
if (limit.ge.1) print *, f1
f2 = 1
if (limit.ge.2) print *, f2
do i = 3, limit
   f3 = f1+f2
   print *, f3
   f1 = f2
   f2 = f3
end do
end
```

6.

```
real x
do
   read *, x
   if (x /= -1.) exit
   print *, 'input value -1. invalid'
end do
print *, x/(1.+x)
end
```

7.

```
type(entry), pointer :: first, current, previous
current => first
if (current%index == 10) then
   first => first%next
else
   do
      previous => current
      current => current%next
      if (current%index == 10) exit
   end do
   previous%next => current%next
end if
```

Chapter 5

1.

```
subroutine calculate(x, n, mean, variance, ok)
   integer, intent(in)             :: n
   real, dimension(n), intent(in) :: x
   real, intent(out)               :: mean, variance
   logical, intent(in)             :: ok
   integer :: i
   mean = 0.
   variance = 0.
   ok = n > 1
   if (ok) then
      do i = 1, n
         mean = mean + x(i)
      end do
      mean = mean/n
      do i = 1, n
         variance = variance + (x(i) - mean)**2
      end do
      variance = variance/(n-1)
   end if
end subroutine calculate
```

Note: A simpler method will become apparent in Chapter 9.

2.

```
subroutine matrix_mult(a, b, c, i, j, k)
   integer, intent(in)               :: i, j, k
   real, dimension(i,j), intent(in)  :: a
   real, dimension(j,k), intent(in)  :: b
   real, dimension(i,k), intent(out) :: c
   integer :: l, m, n
   c(1:i, 1:k) = 0.
   do n = 1, k
      do l = 1, j
         do m = 1, i
            c(m, n) = c(m, n) + a(m, l)*b(l, n)
         end do
      end do
   end do
end subroutine matrix_mult
```

3.

```
subroutine shuffle(cards)
   integer, dimension(52), intent(in) :: cards
   integer                            :: left, choice, i, temp
   real                               :: r
   cards = (/ (i, i=1,52) /)     ! Initialize deck.
   do left = 52,1,-1             ! Loop over number of cards left.
      call random_number(r)      ! Draw a card
      choice = r*left + 1        !    from remaining possibilities
      temp = cards(left)         !    and swap with last
      cards(left) = cards(choice)!    one left.
      cards(choice) = temp
   end do
end subroutine shuffle
```

4.

```
character function earliest(string)
   character(len=*), intent(in) :: string
   integer                      :: j, length
   length = len(string)
   if (length <= 0) then
      earliest = ''
   else
      earliest = string(1:1)
      do j = 2, length
         if (string(j:j) < earliest) earliest = string(j:j)
      end do
   end if
end function earliest
```

5.

```
subroutine sample
   real :: r, l, v, pi
   pi = acos(-1.)
   ⋮
   r = 3.
   l = 4.
   v = volume(r, l)
   ⋮
contains
   function volume(radius, length)
      real, intent(in) :: radius, length
      real             :: volume
      volume = pi*radius**2*length
   end function volume
end subroutine sample
```

7.

```
   module string_type
      type string
         integer :: length
         character(len=80)   :: string_data
      end type string
      interface assignment(=)
         module procedure c_to_s_assign, s_to_c_assign
      end interface (=)
      interface len
         module procedure string_len
      end interface
      interface operator(//)
         module procedure string_concat
      end interface (//)
   contains
      subroutine c_to_s_assign(s, c)
         type (string), intent(out)   :: s
         character(len=*), intent(in) :: c
         s%string_data = c
         s%length = len(c)
         if (s%length > 80) s%length = 80
      end subroutine c_to_s_assign
      subroutine s_to_c_assign(c, s)
         type (string), intent(in)     :: s
         character(len=*), intent(out) :: c
         c = s%string_data(1:s%length)
      end subroutine s_to_c_assign
      function string_len(s)
         integer      :: string_len
         type(string) :: s
         string_len = s%length
      end function string_len
```

```
    function string_concat(s1, s2)
       type (string), intent(in) :: s1, s2
       type (string)             :: string_concat
       string_concat%string_data =                  &
                    s1%string_data(1:s1%length) // &
                    s2%string_data(1:s2%length)
       string_concat%length = s1%length + s2%length
       if (string_concat%length > 80)               &
                    string_concat%length = 80
    end function string_concat
 end module string_type
```

Note: The intrinsic `len` function, used in `subroutine c_to_s_assign`, is described in Section 9.7.1.

Chapter 6

1.

```
type(real_polynomial) a
allocate (a%coeff(4))
a%index = 1
a%coeff = (/ 1, 2, 3, 4 /)
a = real_polynomial(2, (/a%coeff, 5, 6 /))
```

2.

```
type(emfield) a, temp
allocate (a%strength(4, 6))
a%strength = 1.0
temp = a                   ! automatic allocation of temp%content
deallocate (a%strength)
allocate (a%strength(0:5, 0:8))
a%strength(1:4, 1:6) = temp%strength
a%strength(0:5:5, :) = 0
a%strength(1:4, 0) = 0
a%strength(1:4, 7:8) = 0
```

3.

```
type(emfield) a
allocate (a%strength(4, 6))
a%strength = 1.0
a = emfield(reshape( (/ (a%strength(:,i),0.,0.,i=1,6),      &
                        (0.,0.,0.,0.,0.,0.,0.,i=7,9) /), &
                        (/ 6,9/) ))
```

4.

```
b = reshape( [ (0,i=1,size(b,1)+2), (0,b(:,i),0,i=1,size(b,2)), &
            (0,i=1,size(b,1)+2) ], [ size(b,1)+2, size(b,2)+2 ] )
```

Chapter 7

1.

i) `a(1, :)`

ii) `a(:, 20)`

iii) `a(2:50:2, 2:20:2)`

iv) `a(50:2:-2, 20:2:-2)`

v) `a(1:0, 1)`

2.

```
where (z.gt.0) z = 2*z
```

3.

```
integer, dimension(16) :: j
```

4.

`w`	explicit-shape
`a, b`	assumed-shape
`d`	pointer

5.

```
real, pointer :: x(:, :, :)
x => tar(2:10:2, 2:20:2, 2:30:2)%du(3)
```

6.

```
ll = ll + ll
ll = mm + nn + n(j:k+1, j:k+1)
```

7.

```
program backwards
   integer                  :: i, j
   integer, dimension(100) :: reverse
   reverse = (/ (i, i=1, 100) /)
   read *, i, j
   reverse(i:j) = reverse(j:i:-1)
end program backwards
```

Chapter 8

1.

i) `integer, dimension(100) :: bin`

ii) `real(selected_real_kind(6, 4)), dimension(0:20, 0:20) :: &`
`iron_temperature`

iii) `logical, dimension(20) :: switches`

iv) `character(len=70), dimension(44) :: page`

2.

The value of the first `i` is 3.1, but may be changed;
the value of the second `i` is 3.1, but may not be changed.

3.

i) `integer, dimension(100) :: i = (/ (0, k=1, 100) /)`

ii) `integer, dimension(100) :: i = (/ (0, 1, k=1, 50) /)`

iii)
```
real, dimension(10, 10) :: x = reshape((/ (1.0, k=1, 100) /), &
                                       (/10, 10/) )
```

iv) `character(len=10) :: string = '0123456789'`

Note: the `reshape` function will be met in Section 9.15.3.

4.

i) `type(person) boss = person('Smith', 48.7, 22)`

ii) a) `type(entry) new = entry(0.0, 0, null())`

b)
```
type(entry) current
data current%value, current%index /1.0, 1/
```

5.

All are constant expressions except for:

iv) because of the real exponent; and

viii) because of the pointer component.

Chapter 9

1.

```
   program qroots        ! Solution of quadratic equation.
!
      real :: a, b, c, d, x1, x2
!
      read (*, *) a, b, c
      write (*, *) ' a = ', a, 'b = ', b, 'c = ', c
      if (a == 0.) then
         if (b /= 0.) then
            write (*, *) ' Linear: x = ', -c/b
         else
            write (*, *) ' No roots!'
         endif
      else
         d = b**2 - 4.*a*c
         if (d < 0.) then
            write (*, *) ' Complex', -b/(2.*a), '+-',      &
                         sqrt(-d)/(2.*a)
         else
            x1 = -(b + sign(sqrt(d), b))/(2.*a)
            x2 = c/(x1*a)
            write (*, *) ' Real roots', x1, x2
         endif
      endif
   end program qroots
```

Historical note: A similar problem was set in one of the first books on Fortran programming – *A FORTRAN Primer*, E. Organick (Addison-Wesley, 1963). It is interesting to compare Organick's solution, written in FORTRAN II, on p. 122 of that book, with the one above. (It is reproduced in the *Encyclopedia of Physical Science and Technology* (Academic Press, 1987), vol. 5, p. 538.)

2.

```
subroutine calculate(x, mean, variance, ok)
   real, intent(in)     :: x(:)
   real, intent(out)    :: mean, variance
   logical, intent(out) :: ok
   ok = size(x) > 1
   if (ok) then
      mean = sum(x)/size(x)
      variance = sum((x-mean)**2)/(size(x)-1)
   end if
end subroutine calculate
```

3.

F p1 and p2 are associated with the same array elements, but in reverse order

T p1 and p2(4:1:-1) are associated with exactly the same array elements, a(3), a(5), a(7), a(9)

4.

5 1 a has bounds 5:10 and a(:) has bounds 1:6

5 1 p1 has bounds 5:10 and p2 has bounds 1:6

1 1 x and y both have bounds 1:6

5.

```
program command_sum
   use iso_fortran_env
   real(selected_real_kind(15)) :: number, sum
   integer                      :: arglen, i, ios, numbers
   character(1024)              :: arg, error
   intrinsic                    :: command_argument_count, get_command_argument
   sum = 0
   numbers = 0
   do i=1, command_argument_count()
      call get_command_argument(i, arg, arglen)
      if (arglen>len(arg)) then
         write (error_unit, *) 'Ignoring extremely long argument number',i
         cycle
      else if (arglen==0) then
         write (error_unit, *) 'Ignoring zero-length argument number',i
      end if
      if (scan(arg(:arglen), '0123456789.eEdD+-')==0) then
         write (error_unit, *) 'Invalid character in: "',arg(:arglen), '"'
         cycle
      end if
      read (arg(:arglen), *, iostat=ios, iomsg=error) number
      if (ios/=0) then
```

```
         write (error_unit, *) 'Error for "', arg(:arglen), '": ', trim(error)
      else
         numbers = numbers + 1
         sum = sum + number
      end if
   end do
   if (numbers==0) then
      print *, 'No numbers found'
   else
      print *, numbers, 'numbers found, the sum is:', sum
   end if
end program
```

6.

```
! A pseudorandom number generator module with the same
! interface as the standard intrinsic one.
module prng
  use iso_fortran_env, only: int32
  private
  integer(int32), parameter :: a = 16807_int32
  integer(int32), parameter :: m = 2147483647_int32
  integer(int32), parameter :: q = m/a
  integer(int32), parameter :: r = mod(m, a)
  integer :: seed
  integer :: init_count = 0
  public :: random_number, random_seed
  interface random_number
    module procedure random_number_r, random_number_d
  end interface
  interface random_seed
    module procedure random_seed_specific
  end interface
contains
  subroutine init_random_number
    integer :: values(8)
    call date_and_time(values=values)
    seed = init_count + sum(values)
    if (seed<=0 .or. seed>m) seed = 25058 ! Must be in range 1-m.
    init_count = init_count + 1
  end subroutine
  subroutine advance_generator
    integer(int32) :: hi, lo, test
    if (init_count==0) call init_random_number
    !
    ! Calculate seed = mod(a*seed, m),
    ! without overflow or higher-precision arithmetic.
    hi = seed/q
    lo = mod(seed, q)
    test = a*lo - r*hi
```

```
    seed = merge(test, test+m, test>0)
  end subroutine
  impure elemental subroutine random_number_r(harvest)
    real, intent(out) :: harvest
    call advance_generator
    !
    ! Multiply by the reciprocal of m,
    ! to put the result in the range (0.0,1.0).
    !
    harvest = seed*(1.0/m)
  end subroutine
  impure elemental subroutine random_number_d(harvest)
    double precision, intent(out) :: harvest
    call advance_generator
    !
    ! Multiply by the reciprocal of m,
    ! to put the result in the range (0.0,1.0).
    !
    harvest = seed*(1.0d0/m)
  end subroutine
  subroutine random_seed_specific(size, put, get)
    integer, intent(out), optional :: size
    integer, intent(in), optional  :: put(:)
    integer, intent(out), optional :: get(:)
    if (count([present(size), present(put), present(get)])>1) &
      stop '?Too many arguments to RANDOM_SEED'
    if (present(size)) then
      size = 1
    else if (present(put)) then
      if (ubound(put,1)<1) stop '?RANDOM_NUMBER: PUT is too small'
      seed = sum(put)
      if (seed<=0 .or. seed>m) call init_random_number
    else if (present(get)) then
      if (ubound(get,1)<1) stop '?RANDOM_NUMBER: GET is too small'
      get(1) = seed
      get(2:) = 0
    else
      call init_random_number
    end if
  end subroutine
end module
```

Chapter 10

1.

```
character, dimension(3,3) :: tic_tac_toe
integer                   :: unit
⋮
write (unit, '(t1, 3a2)') tic_tac_toe
```

3.

```
subroutine get_char(unit, c, end_of_file)
   integer, intent(in)    :: unit
   character, intent(out) :: c
   logical, intent(out)   :: end_of_file
   integer :: ios
   end_of_file = .false.
   do
      read (unit, '(a1)', advance='no', iostat=ios, end=10) c
      if (ios == 0) return
   end do
10 c = ' '
   end_of_file = .true.
end subroutine get_char
```

5.

```
program check_file_format
   implicit none
   character :: ch, file*1024, lastch = 'x'
   integer   :: crlf_found = 0, lf_found = 0, ios
   call get_command_argument(1, file)
   open (10, file=file, form='unformatted', access='stream', action='read')
   do
      read (10, iostat=ios) ch
      if (is_iostat_end(ios)) exit
      if (ios/=0) stop 'I/O error'
      if (ch==achar(10)) then
         if (lastch==achar(13)) then
            crlf_found = crlf_found + 1
         else
            lf_found = lf_found + 1
         end if
      end if
      lastch = ch
   end do
   if (lf_found>0) print *, lf_found, 'Unix record terminators'
   if (crlf_found>0) print *, crlf_found, 'DOS/Windows record terminators'
   if (lf_found+crlf_found==0) print *, 'No record terminators found'
end program
```

Chapter 11

1.

i) `print '(a/ (t1, 10f6.1))', ' grid', grid`

ii) `print '(a, " ", 25i5)', ' list', (list(i), i = 1, 49, 2)`
or
`print '(a, " ", 25i5)', ' list', list(1:49:2)`

iii) `print '(a/ (" ", 2a12))', ' titles', titles`

iv) `print '(a/ (t1, 5en15.6))', ' power', power`

v) `print '(a, 10l2)', ' flags', flags`

vi) `print '(a, 5(" (", 2f6.1, ")"))', ' plane', plane`

2.

i)
```
read (*, *) grid
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
```

ii)
```
read (*, *) list(1:49:2)
25*1
```

iii)
```
read (*, *) titles
data transfer
```

iv)
```
read (*, *) power
1.0 1.e-03
```

v)
```
read (*, *) flags
t f t f t f f t f t
```

vi)
```
read (*, *) plane
(0.0, 1.0),(2.3, 4)
```

3.

```
! Function to format a 15+ digit real value as if it were Euros.
! In particular, we expect a decimal comma, and no negative zero.
!
! The function returns an allocatable deferred-length string.
!
function real_to_estring(value) result(r)
    real(kind=selected_real_kind(15)), intent(in) :: value
    character(:), allocatable :: r
    character(15) :: format
    character(precision(value)+2) :: temp
    !
    ! Set the size of the format field so that it will be filled
    ! with asterisks if the magnitude of the value is such that the
    ! "cent" field is beyond the decimal precision.
    !
    write (format, '("(ss, dc, f", i0, ".2)")') precision(value) + 1
    write (temp, format) abs(value)
    if (value<0) then
        r = '-'//trim(adjustl(temp))
    else
        r = trim(adjustl(temp))
    end if
end function
```

4.

```
program show_sign_effects
   write (*, 10)                  'default', 1., 2., 3., 4.
   write (*, 10, sign='suppress') 'suppress', 1., 2., 3., 4.
   write (*, 10, sign='plus')     'plus', 1., 2., 3., 4.
10 format (1x, a10, ': ', f6.2, ss, f6.2, sp, f6.2, s, f6.2)
end program
```

```
 default:   1.00  2.00 +3.00  4.00
suppress:   1.00  2.00 +3.00  4.00
    plus:  +1.00  2.00 +3.00  4.00
```

Chapter 13

1.

```
type cmplx (kind)
   integer       :: kind = kind(0.0)
   real, private :: r, theta
end type cmplx
```

2.

```
function cat(a, b)
    type(char_with_max_length(*)), intent(in) :: a, b
    type(char_with_max_length(a%len+b%len))   :: cat
    cat%value = a%value(1:a%len)//b%value(1:b%len)
    cat%len = len(cat%value)
end function cat
integer function indx(string, substring, back)
    type (char_with_max_length(*)), intent(in) :: string, substring
    logical, optional                          :: back
    indx = index(string%value(1:string%len), &
                 substring%value(1:substring%len), back)
end function indx
```

Chapter 14

1.

```
! Interface for an error-handling callback procedure.
abstract interface
   logical function handler_proc(error_code, error_msg)
      integer, intent(in)      :: error_code
      character(*), intent(in) :: error_msg
      ! Return .false to terminate execution
      ! (perhaps after further diagnostics).
      ! Return .true. to attempt to continue processing.
   end function
end interface
```

```
type error_handling
   logical                          :: stop_on_unhandled_error = .True.
   integer                          :: error_code = .False.
   character(:), allocatable        :: error_msg
   procedure(handler_proc), pointer :: error_handler => Null()
end type
```

Chapter 15

3.

```
! Module containing a vector type that counts the number of accesses,
! both as a whole vector and by element.
module counting_vector
   use iso_fortran_env, only: int64
   private
   ! All module entities are private except for this type.
   type, public :: realvec_t
      private
      real, pointer  :: value(:)
      logical        :: allocated = .false.
      integer(int64) :: ecount = 0, vcount = 0
   contains
      procedure :: element, get_usage, new, vector
      final :: zap
   end type

contains
   function element(vec, sub)
      class(realvec_t), intent(inout) :: vec
      integer, intent(in)             :: sub
      real, pointer                   :: element
      vec%ecount = vec%ecount + 1
      element => vec%value(sub)
   end function
   subroutine new(this, n)
      class(realvec_t), intent(inout) :: this
      integer, intent(in)             :: n
      if (this%allocated) deallocate (this%value)
      allocate (this%value(n))
     this%allocated = .true.
   end subroutine
   function vector(vec)
      class(realvec_t), intent(inout) :: vec
      real, pointer                   :: vector(:)
      vec%vcount = vec%vcount + 1
      vector => vec%value
   end function
```

```
   subroutine get_usage(this, ecount, vcount)
      class(realvec_t), intent(in)           :: this
      integer(int64), intent(out), optional  :: ecount, vcount
      if (present(ecount)) ecount = this%ecount
      if (present(vcount)) vcount = this%vcount
   end subroutine
   elemental subroutine zap(this)
      type(realvec_t), intent(inout) :: this
      if (this%allocated) deallocate (this%value)
      this%allocated = .false.
   end subroutine
end module
⋮
type(realvec_t) :: a
integer(int64)  :: nerefs
integer         :: i
⋮
call a%new(n)
do i=1, n
   a%element(i) = ...
end do
print *,a%vector()
⋮
call a%get_usage(ecount=nerefs)
```

Chapter 17

1.

```
program main
   implicit none
   real :: z[*]
   integer :: image
   if (this_image()==1) then
      open (10, file='ex1.data')
      read (10, *) z
      do image = 2, num_images()
         z[image] = z
      end do
   end if
   sync all
   write (*, '(a,f6.3,a,i2)') 'z=', z, ' on image', this_image()
end program
```

Without a `sync all` statement, an image might attempt to write its value before image 1 has set it.

2.

```
program main
   implicit none
   real, allocatable :: z(:)[:]
   integer :: image
   allocate (z(3)[*])
   if (this_image()==1) then
      z(:) = [1.2, 1.3, 1.4]
      do image = 2, num_images()
         z(:)[image] = z(:) + image - 1
      end do
   end if
   sync all
   write (*, '(a,3f4.1,a,i2)') 'z=', z, ' on image', this_image()
end program main
```

Synchronization is built into the `allocate` statement for a coarray, so a `sync all` statement is not needed.

3.

```
subroutine collective_add(a)
   real       :: a[*]
   real, save :: b[*]
   integer    :: i, me, ne, you
   me = this_image()
   ne = num_images()
   i = 1
   do
      sync all
      if (i>=ne) exit
      you = me + i
      if (you>ne) you = you - ne
      b = a + a[you]
      sync all
      a = b
      i = i*2
   end do
end subroutine
```

4.

```
subroutine laplace(nrow, ncol, u)
   integer, intent(in)  ::  nrow, ncol
   real, intent(inout)  ::  u(nrow)[*]
   real                 ::  new_u(nrow)
   integer              ::  i, me, left, right
   me = this_image()
   left = merge(ncol, me-1, me==1)
   right = merge(1, me+1, me==ncol)
   new_u(1) = u(nrow) + u(2)
   new_u(nrow) = u(1) + u(nrow-1)
   new_u(2:nrow-1) = u(1:nrow-2) + u(3:nrow)
   new_u(1:nrow) = new_u(1:nrow) + u(1:nrow)[left] + u(1:nrow)[right]
   sync all
   u(1:nrow) = new_u(1:nrow) - 4.0*u(1:nrow)
end subroutine laplace
```

5.

Code with bottlenecks	Code without bottlenecks

`k = this_image()`
`if (k<=nz) then`
`  do i = 1, nx`
`  a(i, 1:ny) = b(1:ny, k)[i]`
`  end do`
`end if`
	`k = this_image()`
	`if (k<=nz) then`
	`  do ii = 1, nx`
	`  i = 1 + mod(k+ii, nx)`
	`  a(i, 1:ny) = b(1:ny, k)[i]`
	`  end do`
	`end if`

6.

```
module teams
contains
   subroutine team_add(team, team_position, a)
      integer    :: team(:), team_position
      real       :: a[*]
      real, save :: b[*]
      integer    :: i, me, ne, you
      me = team_position
      ne = size(team)
      i = 1
      do
         sync images (team)
         if (i>=ne) exit
         you = me + i
         if (you>ne) you = you - ne
         b = a + a[team(you)]
         sync images (team)
         a = b
         i = i*2
      end do
   end subroutine
end module

program main
   use teams
   implicit none
   integer, allocatable :: team(:)
   real                 :: a[*]
   integer              :: i, me, ne, team_position
   me = this_image()
   ne = num_images()
   allocate ( team(ne/2) )
   if (me<=ne/2) then
      team = [ (i,i=1,ne) ]
      team_position = me
   else
      team = [ (i,i=ne/2+1,ne) ]
      team_position = me - ne/2
   end if
   a = this_image()
   sync all
   call team_add(team,team_position, a)
   write(*,'(a,f6.1,a,i2)') 'a=',a,' on image', this_image()
end program main
```

Chapter 19

1.

```
interface erf
   function erff(x) bind(c)
      use iso_c_binding
      real(c_float), value :: x
      real(c_float)        :: erff
   end function
   function erf(x) bind(c)
      use iso_c_binding
      real(c_double), value :: x
      real(c_double)        :: erf
   end function
end interface
```

2.

```
! float dot_productf(float a[], float b[], size_t n);
!
function dot_productf(a, b, n) bind(c)
  use iso_c_binding
  integer(c_size_t), value  :: n
  real(c_float), intent(in) :: a(n), b(n)
  dot_productf = dot_product(a, b)
end function

! double dot_product(double a[], double b[], size_t n);
!
function dot_productd(a, b, n) bind(c, name='dot_product')
  use iso_c_binding
  integer(c_size_t), value   :: n
  real(c_double), intent(in) :: a(n), b(n)
  dot_productd = dot_product(a, b)
end function
```

Index