Modern Fortran
explained

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
http://www.oup.co.uk/academic/science/maths/series/nmsc

* 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 CFD*
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. Böhmer: *Numerical methods for nonlinear elliptic differential equations: a synopsis*
M. Metcalf, J. Reid, and M. Cohen: *Modern Fortran Explained*

Modern Fortran Explained

Michael Metcalf
Formerly of CERN, Geneva, Switzerland

John Reid
JKR Associates, Oxfordshire

and

Malcolm Cohen
The Numerical Algorithms Group, Oxfordshire

OXFORD
UNIVERSITY PRESS

OXFORD
UNIVERSITY PRESS

Great Clarendon Street, Oxford OX2 6DP

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 in

Oxford New York

Auckland Cape Town Dar es Salaam Hong Kong Karachi
Kuala Lumpur Madrid Melbourne Mexico City Nairobi
New Delhi Shanghai Taipei Toronto

With offices in

Argentina Austria Brazil Chile Czech Republic France Greece
Guatemala Hungary Italy Japan Poland Portugal Singapore
South Korea Switzerland Thailand Turkey Ukraine Vietnam

Oxford is a registered trade mark of Oxford University Press
in the UK and in certain other countries

Published in the United States
by Oxford University Press Inc., New York

First edition published 1987 as *Fortran 8x Explained*
Second edition published 1989
Third edition published 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*
This edition published 2011

British Library Cataloguing in Publication Data

Data available

Library of Congress Cataloging in Publication Data

Library of Congress Control Number: 2010941705

Printed in Great Britain
on acid-free paper by
CPI Antony Rowe, Chippenham, Wiltshire

ISBN 978–0–19–960141–7 (Hbk)
978–0–19–960142–4 (Pbk)

1 3 5 7 9 10 8 6 4 2

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. This revision, which included object-oriented programming features, is now known as Fortran 2003. This has now been followed by a further revision, Fortran 2008, and, once again, it seems appropriate to prepare a definitive informal description of the language that it 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 too (1996 and 1999) and *Fortran 95/2003* (2004), with its added chapters on Fortran 2003. In that final endeavour, a third co-author was welcomed.

In this book, an initial chapter sets out the background to the work on new standards, and the nine following chapters describe Fortran 95 (less its obsolescent features and the redundant Fortran 77 features whose use we deprecate) in a manner suitable both for grasping the implications of its features, and for writing programs. We include the allocatable array extensions that were originally published as an ISO Technical Report and are now part of Fortran 2003, since they have been implemented in Fortran 95 compilers for many years. 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 Fortran 95 standard) and other features whose use we deprecate. They may be encountered in old programs, but are not needed in new ones.

Chapter 11 describes another part of Fortran 2003 that was originally defined by an ISO Technical Report. This is followed, in Chapters 12 to 17, by descriptions of the other features defined by the Fortran 2003 standard. Chapter 18 describes a part of Fortran 2008 that was originally defined by an ISO Technical Report and two further chapters describe the other new features of Fortran 2008. The structure of the book thus allows the reader to distinguish clearly between Fortran 95 (plus allocatable array extensions), Fortran 2003, and the new Fortran 2008 features. Note that, apart from a small number of deletions, each of the languages Fortran 77, Fortran 90, Fortran 95, Fortran 2003, and Fortran 2008 is a subset of its successor.

In order to make the book a complete reference work, it concludes with seven appendices. They contain, successively, a list of the intrinsic procedures, 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, advice on avoiding compilation cascades, an extended example illustrating the use of object orientation, a glossary of Fortran terms, and solutions to most of the exercises.

It is our hope that this book, by providing complete descriptions of Fortran 95, Fortran 2003 and Fortran 2008, 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.

* * *

Malcolm Cohen wishes to thank the Numerical Algorithms Group (NAG) for its encouragement during the writing of this book.

Conventions used in this book

Fortran displayed text is set in typewriter font:

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

and a line consisting of a colon indicates omitted lines:

```
subroutine sort
   :
end subroutine sort
```

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 files *edits.ps* and *edits.pdf* at `ftp://ftp.numerical.rl.ac.uk/pub/MRandC`.

This page intentionally left blank

Contents

1 Whence Fortran? 1

1.1 Introduction . . . 1
1.2 Fortran's early history . . . 2
1.3 The drive for the Fortran 90 standard . . . 3
1.4 Language evolution . . . 4
1.5 Fortran 95 . . . 4
1.6 Extensions to Fortran 95 . . . 5
1.7 Fortran 2003 . . . 6
1.8 Fortran 2008 . . . 7
1.9 Conformance . . . 7

2 Language elements 9

2.1 Introduction . . . 9
2.2 Fortran character set . . . 9
2.3 Tokens . . . 10
2.4 Source form . . . 11
2.5 Concept of type . . . 13
2.6 Literal constants of intrinsic type . . . 14
2.6.1 Integer literal constants . . . 14
2.6.2 Real literal constants . . . 15
2.6.3 Complex literal constants . . . 17
2.6.4 Character literal constants . . . 17
2.6.5 Logical literal constants . . . 19
2.7 Names . . . 20
2.8 Scalar variables of intrinsic type . . . 20
2.9 Derived data types . . . 21
2.10 Arrays of intrinsic type . . . 23
2.11 Character substrings . . . 26
2.12 Objects and subobjects . . . 27
2.13 Pointers . . . 27
2.14 Summary . . . 29

3 Expressions and assignments 33

3.1 Introduction . . . 33

3.2 Scalar numeric expressions . . . 34
3.3 Defined and undefined variables . . . 37
3.4 Scalar numeric assignment . . . 38
3.5 Scalar relational operators . . . 38
3.6 Scalar logical expressions and assignments . . . 39
3.7 Scalar character expressions and assignments . . . 41
3.8 Structure constructors and scalar defined operators . . . 42
3.9 Scalar defined assignments . . . 45
3.10 Array expressions . . . 46
3.11 Array assignment . . . 48
3.12 Pointers in expressions and assignments . . . 48
3.13 The nullify statement . . . 51
3.14 Summary . . . 51

4 Control constructs 55
4.1 Introduction . . . 55
4.2 The if construct and statement . . . 55
4.3 The case construct . . . 57
4.4 The do construct . . . 59
4.5 The go to statement . . . 62
4.6 Summary . . . 63

5 Program units and procedures 67
5.1 Introduction . . . 67
5.2 Main program . . . 68
5.3 The stop statement . . . 69
5.4 External subprograms . . . 69
5.5 Modules . . . 70
5.6 Internal subprograms . . . 73
5.7 Arguments of procedures . . . 73
5.7.1 Pointer arguments . . . 75
5.7.2 Restrictions on actual arguments . . . 76
5.7.3 Arguments with the target attribute . . . 76
5.8 The return statement . . . 77
5.9 Argument intent . . . 77
5.10 Functions . . . 78
5.10.1 Prohibited side-effects . . . 80
5.11 Explicit and implicit interfaces . . . 80
5.12 Procedures as arguments . . . 82
5.13 Keyword and optional arguments . . . 83
5.14 Scope of labels . . . 85
5.15 Scope of names . . . 85
5.16 Direct recursion . . . 88
5.17 Indirect recursion . . . 89
5.18 Overloading and generic interfaces . . . 90

5.19 Assumed character length . . . 93
5.20 The subroutine and function statements . . . 95
5.21 Summary . . . 96

6 Array features **99**
6.1 Introduction . . . 99
6.2 Zero-sized arrays . . . 99
6.3 Assumed-shape arrays . . . 100
6.4 Automatic objects . . . 100
6.5 Allocation of data . . . 102
6.5.1 The allocatable attribute . . . 102
6.5.2 The allocate statement . . . 103
6.5.3 The deallocate statement . . . 104
6.5.4 Allocatable dummy arguments . . . 105
6.5.5 Allocatable functions . . . 105
6.5.6 Allocatable components . . . 106
6.5.7 Allocatable arrays vs. pointers . . . 109
6.6 Elemental operations and assignments . . . 110
6.7 Array-valued functions . . . 110
6.8 The where statement and construct . . . 111
6.9 The forall statement and construct . . . 114
6.10 Pure procedures . . . 117
6.11 Elemental procedures . . . 118
6.12 Array elements . . . 119
6.13 Array subobjects . . . 120
6.14 Arrays of pointers . . . 123
6.15 Pointers as aliases . . . 124
6.16 Array constructors . . . 125
6.17 Mask arrays . . . 126
6.18 Summary . . . 127

7 Specification statements **133**
7.1 Introduction . . . 133
7.2 Implicit typing . . . 133
7.3 Declaring entities of differing shapes . . . 134
7.4 Named constants and constant expressions . . . 134
7.5 Initial values for variables . . . 137
7.5.1 Initialization in type declaration statements . . . 137
7.5.2 The data statement . . . 138
7.5.3 Pointer initialization and the function null . . . 140
7.5.4 Default initialization of components . . . 141
7.6 The public and private attributes . . . 142
7.7 The pointer, target, and allocatable statements . . . 144
7.8 The intent and optional statements . . . 144
7.9 The save attribute . . . 145

7.10 The use statement . . . 146
7.11 Derived-type definitions . . . 149
7.12 The type declaration statement . . . 150
7.13 Type and type parameter specification . . . 152
7.14 Specification expressions . . . 153
7.14.1 Specification functions . . . 153
7.15 The namelist statement . . . 155
7.16 Summary . . . 156

8 Intrinsic procedures **161**
8.1 Introduction . . . 161
8.1.1 Keyword calls . . . 161
8.1.2 Categories of intrinsic procedures . . . 162
8.1.3 The intrinsic statement . . . 162
8.1.4 Argument intents . . . 162
8.2 Inquiry functions for any type . . . 162
8.3 Elemental numeric functions . . . 163
8.3.1 Elemental functions that may convert . . . 163
8.3.2 Elemental functions that do not convert . . . 164
8.4 Elemental mathematical functions . . . 165
8.5 Elemental character and logical functions . . . 166
8.5.1 Character–integer conversions . . . 166
8.5.2 Lexical comparison functions . . . 167
8.5.3 String-handling elemental functions . . . 167
8.5.4 Logical conversion . . . 168
8.6 Non-elemental string-handling functions . . . 168
8.6.1 String-handling inquiry function . . . 168
8.6.2 String-handling transformational functions . . . 168
8.7 Numeric inquiry and manipulation functions . . . 168
8.7.1 Models for integer and real data . . . 168
8.7.2 Numeric inquiry functions . . . 169
8.7.3 Elemental functions to manipulate reals . . . 170
8.7.4 Transformational functions for kind values . . . 171
8.8 Bit manipulation procedures . . . 171
8.8.1 Inquiry function . . . 172
8.8.2 Elemental functions . . . 172
8.8.3 Elemental subroutine . . . 173
8.9 Transfer function . . . 173
8.10 Vector and matrix multiplication functions . . . 174
8.11 Transformational functions that reduce arrays . . . 175
8.11.1 Single argument case . . . 175
8.11.2 Optional argument dim . . . 175
8.11.3 Optional argument mask . . . 176
8.12 Array inquiry functions . . . 176
8.12.1 Allocation status . . . 176

8.12.2 Bounds, shape, and size . . . 176
8.13 Array construction and manipulation functions . . . 177
8.13.1 The merge elemental function . . . 177
8.13.2 Packing and unpacking arrays . . . 177
8.13.3 Reshaping an array . . . 177
8.13.4 Transformational function for replication . . . 178
8.13.5 Array shifting functions . . . 178
8.13.6 Matrix transpose . . . 179
8.14 Transformational functions for geometric location . . . 179
8.15 Transformational function for pointer disassociation . . . 179
8.16 Non-elemental intrinsic subroutines . . . 180
8.16.1 Real-time clock . . . 180
8.16.2 CPU time . . . 181
8.16.3 Random numbers . . . 181
8.17 Summary . . . 182

9 Data transfer **185**
9.1 Introduction . . . 185
9.2 Number conversion . . . 185
9.3 I/O lists . . . 186
9.4 Format definition . . . 188
9.5 Unit numbers . . . 190
9.6 Internal files . . . 191
9.7 Formatted input . . . 193
9.8 Formatted output . . . 194
9.9 List-directed I/O . . . 195
9.10 Namelist I/O . . . 197
9.11 Non-advancing I/O . . . 198
9.12 Edit descriptors . . . 200
9.12.1 Repeat counts . . . 200
9.12.2 Data edit descriptors . . . 201
9.12.3 Character string edit descriptor . . . 205
9.12.4 Control edit descriptors . . . 205
9.13 Unformatted I/O . . . 208
9.14 Direct-access files . . . 209
9.15 Execution of a data transfer statement . . . 210
9.16 Summary . . . 211

10 Operations on external files **213**
10.1 Introduction . . . 213
10.2 Positioning statements for sequential files . . . 214
10.2.1 The backspace statement . . . 214
10.2.2 The rewind statement . . . 214
10.2.3 The endfile statement . . . 215
10.2.4 Data transfer statements . . . 215

10.3 The open statement 216
10.4 The close statement 218
10.5 The inquire statement 219
10.6 Summary 222

11 Floating-point exception handling 223
11.1 Introduction 223
11.2 The IEEE standard 224
11.3 Access to the features 225
11.4 The Fortran flags 227
11.5 Halting 228
11.6 The rounding mode 228
11.7 The underflow mode (Fortran 2003 only) 229
11.8 The module ieee_exceptions 229
11.8.1 Derived types 229
11.8.2 Inquiry functions for IEEE exceptions 230
11.8.3 Subroutines for the flags and halting modes 230
11.8.4 Subroutines for the whole of the floating-point status 231
11.9 The module ieee_arithmetic 232
11.9.1 Derived types 232
11.9.2 Inquiry functions for IEEE arithmetic 232
11.9.3 Elemental functions 234
11.9.4 Non-elemental subroutines 235
11.9.5 Transformational function for kind value 236
11.10 Examples 237
11.10.1 Dot product 237
11.10.2 Calling alternative procedures 237
11.10.3 Calling alternative in-line code 238
11.10.4 Reliable hypotenuse function 238
11.10.5 Access to IEEE arithmetic values 239

12 Interoperability with C 243
12.1 Introduction 243
12.2 Interoperability of intrinsic types 243
12.3 Interoperability with C pointer types 245
12.4 Interoperability of derived types 246
12.5 Interoperability of variables 247
12.6 The value attribute 248
12.7 Interoperability of procedures 249
12.8 Interoperability of global data 250
12.9 Invoking a C function from Fortran 251
12.10 Invoking Fortran from C 252
12.11 Enumerations 253

13 Type parameters and procedure pointers **255**
13.1 Introduction 255
13.2 Deferred type parameters 255
13.3 Type parameter enquiry 256
13.4 Parameterized derived types 256
13.4.1 Defining a parameterized derived type 256
13.4.2 Assumed and deferred type parameters 258
13.4.3 Default type parameter values 258
13.4.4 Derived type parameter enquiry 259
13.5 Abstract interfaces 259
13.6 Procedure pointers 261
13.6.1 Procedure pointer variables 261
13.6.2 Procedure pointer components 261
13.6.3 The pass attribute 262

14 Object-oriented programming **265**
14.1 Introduction 265
14.2 Type extension 265
14.2.1 Type extension and type parameters 267
14.3 Polymorphic entities 267
14.3.1 Establishing the dynamic type 268
14.3.2 Limitations on the use of a polymorphic variable 269
14.3.3 Polymorphic arrays and scalars 269
14.3.4 Unlimited polymorphic entities 269
14.3.5 Polymorphic entities and generic resolution 270
14.4 The associate construct 271
14.5 The select type construct 272
14.6 Type-bound procedures 274
14.6.1 Specific type-bound procedures 274
14.6.2 Generic type-bound procedures 277
14.6.3 Type extension and type-bound procedures 279
14.7 Deferred bindings and abstract types 280
14.8 Finalization 281
14.8.1 Type extension and final subroutines 284
14.9 Procedure encapsulation example 284
14.10 Type inquiry functions 286

15 Establishing and moving data **289**
15.1 Introduction 289
15.2 Mixed component accessibility 289
15.3 Structure constructors 289
15.4 The allocate statement 291
15.4.1 Typed allocation and deferred type parameters 291
15.4.2 Polymorphic variables and typed allocation 292
15.4.3 Sourced allocation 292

15.5 Allocatable entities 293
15.5.1 Allocatable scalars 294
15.5.2 Assignment to an allocatable array 294
15.5.3 Transferring an allocation 295
15.6 Pointer assignment 296
15.7 More control of access from a module 296
15.8 Renaming operators on the use statement 297
15.9 Array constructor syntax 297
15.10 Specification and constant expressions 298

16 Miscellaneous enhancements **301**
16.1 Introduction 301
16.2 Pointer intent 301
16.3 The volatile attribute 301
16.3.1 Volatile semantics 302
16.3.2 Volatile scoping 303
16.3.3 Volatile arguments 304
16.4 The import statement 304
16.5 Intrinsic modules 306
16.6 Access to the computing environment 307
16.6.1 Environment variables 307
16.6.2 Information about the program invocation 308
16.7 Support for internationalization 308
16.7.1 Character sets 309
16.7.2 ASCII character set 309
16.7.3 ISO 10646 character set 310
16.7.4 UTF-8 files 310
16.7.5 Decimal comma for input/output 311
16.8 Lengths of names and statements 312
16.9 Binary, octal, and hexadecimal constants 312
16.10 Other changes to intrinsic procedures 313
16.11 Error message retrieval 314
16.12 Enhanced complex constants 314
16.13 Interface block extensions 314
16.14 Public entities of private type 315

17 Input/output enhancements **317**
17.1 Introduction 317
17.2 Non-default derived-type input/output 317
17.3 Asynchronous input/output 320
17.4 The asynchronous attribute 322
17.5 Input and output of IEEE exceptional values 323
17.6 Stream access input/output 323
17.7 Recursive input/output 324
17.8 The flush statement 324

17.9 Comma after a P edit descriptor . . . 324
17.10 The iomsg= specifier . . . 325
17.11 The round= specifier . . . 325
17.12 The sign= specifier . . . 325
17.13 Kind type parameters of integer and logical specifiers . . . 325
17.14 More specifiers in read and write statements . . . 326
17.15 Intrinsic functions for I/O status testing . . . 326
17.16 Some inquire statement enhancements . . . 326
17.17 Namelist enhancements . . . 327

18 Enhanced module facilities 329

18.1 Introduction . . . 329
18.2 Submodules . . . 330
18.2.1 Separate module procedures . . . 330
18.2.2 Submodules of submodules . . . 331
18.2.3 Submodule entities . . . 331
18.2.4 Submodules and use association . . . 332
18.3 The advantages of submodules . . . 332

19 Coarrays 333

19.1 Introduction . . . 333
19.2 Referencing images . . . 334
19.3 The properties of coarrays . . . 335
19.4 Accessing coarrays . . . 336
19.5 The sync all statement . . . 337
19.6 Coarrays in procedures . . . 338
19.7 Allocatable coarrays . . . 340
19.8 Coarrays with allocatable or pointer components . . . 341
19.8.1 Data components . . . 341
19.8.2 Procedure pointer components . . . 342
19.9 Coarray components . . . 342
19.10 References to polymorphic subobjects . . . 343
19.11 Volatile and asynchronous attributes . . . 343
19.12 Interoperability . . . 343
19.13 Synchronization . . . 343
19.13.1 Execution segments . . . 343
19.13.2 The sync images statement . . . 344
19.13.3 The lock and unlock statements . . . 345
19.13.4 Critical sections . . . 347
19.13.5 The sync memory statement and atomic subroutines . . . 347
19.13.6 The stat= and errmsg= specifiers in synchronization statements . . . 348
19.13.7 The image control statements . . . 348
19.14 Program termination . . . 348
19.15 Input/output . . . 349
19.16 Intrinsic procedures . . . 351

19.16.1 Inquiry functions . . . 351
19.16.2 Transformational functions . . . 351

20 Other Fortran 2008 enhancements **353**
20.1 Trivial syntactic conveniences . . . 353
20.1.1 Implied-shape arrays . . . 353
20.1.2 Implied-do loops in data statements . . . 353
20.1.3 Type-bound procedures . . . 354
20.1.4 Structure constructors . . . 354
20.1.5 Semicolons . . . 355
20.1.6 The stop statement . . . 355
20.1.7 Exit from nearly any construct . . . 355
20.2 Limitation changes . . . 356
20.2.1 64-bit integer support . . . 356
20.2.2 Maximum array rank . . . 356
20.3 Data expressiveness . . . 356
20.3.1 Allocatable components of recursive type . . . 356
20.3.2 Initial pointer association . . . 358
20.4 Performance-oriented features . . . 359
20.4.1 The do concurrent construct . . . 359
20.4.2 The contiguous attribute . . . 361
20.4.3 Simply contiguous array designators . . . 364
20.5 Computational expressiveness . . . 365
20.5.1 Accessing parts of complex variables . . . 365
20.5.2 Pointer functions denoting variables . . . 366
20.5.3 The block construct . . . 366
20.5.4 Impure elemental procedures . . . 368
20.5.5 Internal procedures as actual arguments . . . 370
20.5.6 Specifying the kind of a forall index variable . . . 370
20.5.7 Generic resolution . . . 371
20.6 Data usage and computation . . . 372
20.6.1 Enhancements to the allocate statement . . . 372
20.6.2 Automatic reallocation . . . 373
20.6.3 Elemental subprogram restrictions . . . 373
20.7 Input/output . . . 374
20.7.1 Recursive input/output . . . 374
20.7.2 The newunit= specifier . . . 374
20.7.3 Writing comma-separated values . . . 375
20.8 Intrinsic procedures . . . 376
20.9 Mathematical intrinsic functions . . . 376
20.9.1 Changes to trigonometric functions . . . 376
20.9.2 New hyperbolic trigonometic functions . . . 376
20.9.3 New special mathematical functions . . . 377
20.9.4 Euclidean norms . . . 378
20.10 Bit manipulation . . . 378

20.10.1 Bitwise (unsigned) comparison . . . 378
20.10.2 Double-width shifting . . . 379
20.10.3 Bitwise reductions . . . 379
20.10.4 Counting bits . . . 380
20.10.5 Producing bitmasks . . . 380
20.10.6 Merging bits . . . 381
20.10.7 Additional shift operations . . . 381
20.11 Miscellaneous intrinsic procedures . . . 382
20.11.1 Procedures supporting coarrays . . . 382
20.11.2 Executing another program . . . 382
20.11.3 Character comparison . . . 383
20.11.4 Array searching . . . 383
20.11.5 Logical parity . . . 383
20.11.6 Decimal arithmetic support . . . 384
20.11.7 Size of an object in memory . . . 384
20.12 Additions to the iso_fortran_env module . . . 385
20.12.1 Compilation information . . . 385
20.12.2 Names for common kinds . . . 385
20.12.3 Kind arrays . . . 386
20.12.4 Coarray support facilities . . . 386
20.13 Changes to other standard intrinsic modules . . . 387
20.13.1 The iso_c_binding module . . . 387
20.13.2 The ieee_arithmetic module . . . 387
20.14 Programs and procedures . . . 388
20.14.1 Saved module entities . . . 388
20.14.2 Automatic pointer targetting . . . 388
20.14.3 Denoting absent arguments . . . 389

A Intrinsic procedures **393**

B Deprecated features **399**
B.1 Introduction . . . 399
B.2 Storage association . . . 399
B.2.1 Storage units . . . 399
B.2.2 The equivalence statement . . . 400
B.2.3 The common block . . . 402
B.2.4 The block data program unit . . . 404
B.2.5 Coarrays and storage association . . . 405
B.3 Shape and character length disagreement . . . 405
B.4 The include line . . . 407
B.5 Other forms of loop control . . . 407
B.5.1 The labelled do construct . . . 407
B.5.2 The do while . . . 408
B.6 Double precision real . . . 408
B.7 The dimension, codimension, and parameter statements . . . 409

B.8 Specific names of intrinsic procedures . . . 410
B.9 Non-default mapping for implicit typing . . . 412
B.10 Fortran 2008 deprecated features . . . 413
B.10.1 The sync memory statement and atomic subroutines . . . 413
B.10.2 Components of type `c_ptr` or `c_funptr` . . . 416
B.10.3 Type declarations . . . 416
B.10.4 Redundant contains statement . . . 417
B.10.5 The end statement . . . 417
B.10.6 Referencing `atan2` by the name `atan` . . . 418

C Obsolescent features **419**
C.1 Obsolescent in Fortran 95 . . . 419
C.1.1 Fixed source form . . . 419
C.1.2 Computed go to . . . 420
C.1.3 Character length specification character`*` . . . 420
C.1.4 Data statements among executables . . . 420
C.1.5 Statement functions . . . 421
C.1.6 Assumed character length of function results . . . 422
C.1.7 Arithmetic if statement . . . 422
C.1.8 Shared do-loop termination . . . 423
C.1.9 Alternate return . . . 423
C.2 Feature obsolescent in Fortran 2008: Entry statement . . . 424
C.3 Feature deleted in Fortran 2003: Carriage control . . . 426
C.4 Features deleted in Fortran 95 . . . 427

D Avoiding compilation cascades **429**

E Object-oriented list example **433**

F Fortran terms **441**

G Solutions to exercises **453**

Index **475**

1. Whence Fortran?

1.1 Introduction

This book is concerned with the Fortran 95, Fortan 2003 and Fortran 2008 programming languages, setting out a reasonably concise description of the whole of each. The form chosen for its presentation is that of a textbook intended for use in teaching or learning the language.

The description of Fortran 95 occupies Chapters 2 to 10 and Appendices B and C. We include the allocatable array extensions that were originally published as an ISO Technical Report, since they have been implemented in Fortran 95 compilers for many years and are now part of Fortran 2003. These chapters are written in such a way that simple programs can already 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, Chapter 6 completes the description of the powerful array features, Chapter 7 considers the details of specifying data objects and derived types, and Chapter 8 details the intrinsic procedures. Chapters 9 and 10 cover the whole of 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. In Appendices B and C, we describe features that are redundant in the language. Those of Appendix B are still fully part of the standard but their use is deprecated by us, while those of Appendix C are designated as obsolescent by the standard.

Chapter 11 describes an official extension to Fortran 95.

Fortran 2003 contains all of Fortran 95, including the extensions of Chapter 11, and Chapters 12 to 18 describe its additional features. Chapter 12 deals with interoperability with the C programming language, Chapter 13 with parameterized data types and procedure pointers, Chapter 14 with object-oriented programming, and Chapter 15 with establishing and manipulating data. Chapter 16 covers some miscellaneous enhancements, while Chapter 17 deals with enhancements in the area of input/output and Chapter 18 with submodules, formally an extension.

Fortran 2008, in its turn, contains the whole of Fortran 2003, including the extensions of Chapter 18, with an addition, coarrays, that is important for parallel processing, as well as a number of lesser enhancements. These are described in Chapters 19 and 20, respectively.

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 fifty 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 1.8 consider the development of Fortran 95 and its extensions, then of Fortran 2003 and Fortran 2008. The final section considers the requirements on programs and processors for conformance with the standard.

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, so-called *octal code*. 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. (In fact, the name Fortran is a contraction of Formula Translation.) This enabled working programs to be written faster 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 were able to concentrate more on the problem in hand. Perhaps more important, however, was the fact that 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.

Fortran brought with it several other advances. It was, for instance, a language which remained close to, and exploited, the available hardware rather than being an abstract concept. It also brought with it the possibility for programmers to control storage allocation in a simple way, a feature which was very necessary in those early days of small memories.

The proliferation of dialects remained a problem after the publication of the 1966 standard. There was a widespread implementation in compilers of features which were essential for large-scale programs, but which were ignored by the standard. Different compilers implemented such facilities 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 pre-processors.

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, formerly known as Fortran 8x and now as Fortran 90. We will use the abbreviations J3 and WG5 for these two committees.

J3 itself is a body 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 that a draft standard become an international standard. 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 which 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 codes, 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 abstract data types. 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. The latter 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 – `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 but was not for Fortran 95.

The second list contains the *obsolescent features*, those considered to be outmoded and redundant, and which are candidates for deletion in the next revision. The Fortran 95 obsolescent features are described in Appendix C.

For Fortran 2003, there were no new obsolescent features and none of the Fortran 95 obsolescent features have been deleted. In Fortran 2008, the `entry` statement has been made obsolescent.

The obsolescent features that were deleted from Fortran 95 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, such that it would be 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 of 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 did become necessary also 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 are 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.

Fortran 95 was backwards compatible with Fortran 90, apart from a minor change in the definition of `sign` (Section 8.3.2) and the deletion of some Fortran 77 features declared obsolete in Fortran 90. However, there were two new intrinsic procedures, `null` and `cpu_time`, which might also be names of external procedures in an existing Fortran 90 program.

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.

1.6 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 these three features:

i) handling floating-point exceptions;

ii) permitting allocatable arrays as structure components, dummy arguments, and function results; and

iii) 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 integrated into the next revision of the Fortran 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

[1] *The High Performance Fortran Handbook*, C. Koebel *et al.*, MIT Press, Cambridge, MA, 1994.
[2] `ftp://ftp.nag.co.uk/sc22wg5/ISO_VARYING_STRING/`

were completed and the first is the subject of Chapter 11. The details of the second have been incorporated into the earlier chapters as it was widely implemented in Fortran 95 compilers. Difficulties were encountered with the third, so the report mechanism was abandoned for interoperability with C, but it was subsequently included in Fortran 2003 (see Chapter 13).

Another auxiliary standard, ISO/IEC 1539-3 : 1999(E), was developed to meet the need of programmers to maintain several versions of code to allow for different systems and different applications. Keeping several copies of the source code is error prone. It is far better to maintain a master code from which any of the versions may be selected. This standard is for a very simple form of conditional compilation, which selects some of the Fortran lines from the source and omits the rest or converts them to comments. The process is controlled by 'coco lines' in the source that are also omitted or converted to comments. This auxilliary standard has met with little success.

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. It is the subject of the middle part of this book. 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.
- 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 4-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.

Fortran 2003 has been slow to be fully implemented in compilers. The standard was augmented by a further Technical Report, published in February 2005, that defines how the use of modules can be enhanced by the use of submodules (see Chapter 12).

1.8 Fortran 2008

Notwithstanding the fact that Fortran 2003-compliant compilers have been very slow to appear, the standardization committees have thought fit to plunge on with yet another standard, Fortran 2008. Its single most important new feature is intended for parallel processing – the addition of coarray handling facilities. Further, the `do concurrent` form of loop control and the `contiguous` attribute are introduced, and other new features include: the submodule extension to Fortran 2003, 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.

1.9 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 from the old one. The syntax rules are expressed in a form of BNF 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:

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 compile-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 95, the basic elements, or character set, are the 26 letters of the English alphabet, the 10 Arabic numerals, 0 to 9, the underscore, _, and the so-called special characters listed in Table 2.1. Fortran 95 does not require the support of lower-case letters, but almost all computers nowadays support them.[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 character sequences. 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, to be £ in the United Kingdom), the graphics are fixed, though their styles are not fixed. The special characters $ and ? have no specific meaning within the Fortran language.[2]

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

[1] Fortran 2003 requires the support of lower-case letters.

[2] Of the additional special characters of Fortran 2003, only square brackets have a specific meaning.

Table 2.1. The special characters of the Fortran language.

	Fortran 95		**Fortran 95**		**Fortran 2003**
=	Equals sign	:	Colon	\	Backslash
+	Plus sign		Blank	[	Left square bracket
-	Minus sign	!	Exclamation mark	]	Right square bracket
*	Asterisk	"	Quotation mark	{	Left curly bracket
/	Slash	%	Percent	}	Right curly bracket
(	Left parenthesis	&	Ampersand	~	Tilde
)	Right parenthesis	;	Semicolon	`	Grave accent
,	Comma	<	Less than	^	Circumflex accent
.	Decimal point	>	Greater than	|	Vertical line
$	Currency symbol	?	Question mark	#	Number sign
'	Apostrophe			@	Commercial at

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 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 3.8), and *separators*, which are[3]

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

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`,

[3]In Fortran 2003, the characters `[` and `]` are also separators.

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.

all stop**	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*

* Fortran 2003 onwards; ** Fortran 2008 only.

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,[4] 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 contains only blanks, or which

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

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 (but not after the final program unit[5]). A *character context* (those contexts defined in Sections 2.6.4 and 9.12.4) 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 39 additional continuation lines are allowed.[6] The so-called *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 `&`. Normally, continuation is 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 since 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 39 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.

When writing short statements one after the other, it can be convenient to write several of them on one line. The semicolon (`;`) character is used as a *statement separator* in these circumstances, for example:

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

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 even long statements one after the other in a solid block of lines, each 132 characters long and

[5]Fortran 2003 allows blank comment lines after the final program unit.

[6]More continuation lines are allowed in Fortran 2003, see Section 16.8.

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 which 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 data 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 data 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 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.[7] 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.

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[8] the range is often from -2147483648 to $+2147483647$.

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 -999999 to 999999 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 -999999 to 999999 with the least margin (see Section 8.7.4).

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

[7] 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 8.7.

[8] Fortran 2008 also requires support for, effectively, a 64-bit integer type, see Section 20.2.1.

```
-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 4-byte integers, and yet support kind values 1, 2, and 4 with this hardware (to ease portability from processors that have hardware for 1-, 2-, and 4-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 8.2):

```
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 8.7.2), as in

```
range(2_k6)
```

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

In addition to the usual integers of the decimal number system, for some applications it is very convenient to be able to represent positive whole numbers in binary, octal, or hexadecimal form. Unsigned constants of these forms exist in Fortran, and are represented as illustrated in these examples:

```
binary (base 2):          b'01100110'
octal (base 8):           o'076543'
hexadecimal (base 16):    z'10fa'
```

In the hexadecimal form, the letters `a` to `f` represent the values beyond 9; they may be used also in upper case. The delimiters may be quotation marks or apostrophes. The use of these forms of constants is limited to their appearance as implicit integers in the `data` statement (Section 7.5.2). 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 9.12.2).[9]

Bits stored as an integer representation may be manipulated by the intrinsic procedures described in Section 8.8.[10]

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

[9] Further possibilities, in Fortran 2003, are described in Section 16.9.

[10] Fortran 2008 has additional bit intrinsic procedures, see Section 20.10.

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 be sure to obtain a desired range and significance 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 B.

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 8.7.2). 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 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`. We 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`;

- `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`;

and, if the lower-case letters are available,

- `a` is less than `b` is less than `c` ... is less than `y` is less than `z`;
- 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 8.5.1), 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.[11]

There is no requirement on 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. 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.

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

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
```

Non-default logical kinds are useful for storing logical arrays compactly; we defer further discussion until Section 6.17.

2.7 Names

A Fortran program references many different entities by name. Such names must consist of between 1 and 31 alphanumeric characters[12] – 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 7.4.

2.8 Scalar variables of intrinsic type

We have seen in the section on literal constants 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
```

[12] Up to 63 characters are allowed in Fortran 2003, see Section 16.8.

```
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. 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 3.8 and 3.9, 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 3.8 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 definition of the type.

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
```

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

Figure 2.1

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

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 10 real elements, we add the `dimension` attribute to the type declaration statement thus:

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

The successive elements of the array 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 seven dimensions[13] to be specified. 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
```

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. 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

[13] Fortran 2008 allows fifteen dimensions.

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
```

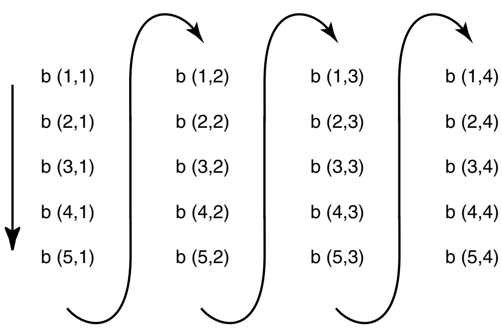
might be used to hold the value of a variable in three dimensions and the values of its first and second derivatives. If `t` is of type `triplet`, `t%du` and `t%d2u` are arrays of type real.

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.

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



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 8.3.1). 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 6.13. 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 an integer array
```

where `ipoint` is an 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 6.13.

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 `(/)`.[14] 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 8.13.3) to reshape a rank-one array constant.

A full description of array constructors is reserved for Section 6.16.

[14] In Fortran 2003, the characters `[` and `]` may be used to delimit an array constructor.

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)
```

At this point we must note a limitation associated with character variables, namely that character variables must have a declared maximum length, making it impossible to manipulate character variables of variable length, unless they are defined appropriately as of a derived data type.[15] Nevertheless, this data type is adequate for most character manipulation applications.

[15] This limitation does not apply in Fortran 2003, see Section 15.2.

2.12 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.

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 object followed by one or more qualifiers (for example, `t(1:7)` and `t(1)%vertex`). Each successive qualifier specifies a part of the 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.

2.13 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 a 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 3.13) 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 in Section 6.5 we will describe a better way to to do this in cases where the 'pointing' property is not essential.

By default, pointers are initially undefined (see also 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 8.15) 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 3.12.

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`.

Pointers will be discussed in detail in later chapters (especially Sections 3.12, 5.7.1, 6.14, 6.15, 7.5.3, 7.5.4, and 8.2).

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
```

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, 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 2.12, 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`.

This page intentionally left blank

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

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 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 8.3.1. In both Tables, I stands for integer, R for real, and C for complex.

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

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 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 of 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.

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 7.5.

In the case of a pointer, the pointer association status may be *undefined*, *associated* with a target, or *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 8.2). 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 7.5.3).

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

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 function `int` is defined in Section 8.3.1).

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*))

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 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

`<`	or `.lt.`	less than
`<=`	or `.le.`	less than or equal
`==`	or `.eq.`	equal
`/=`	or `.ne.`	not equal
`>`	or `.gt.`	greater than
`>=`	or `.ge.`	greater than or equal

If either or both of the expressions are complex, only the operators == and /= (or `.eq.` and `.ne.`) 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 flow 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. 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.

No other form of mixed-mode relational operator is intrinsically available, though such an operator may be defined (Section 3.8). 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
```

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

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 `.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 result of

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

is the string `POLE`.

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 8.5 and 8.6.

3.8 Structure constructors and scalar defined operators

No operators for derived types are automatically available, but 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

type-name (*expr-list*)

where the *expr-list* specifies the values of the components. For example, given the type

```
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)
```

Each expression in *expr-list* corresponds to a component of the structure; if it is not a pointer component, the value is assigned to the component under the rules of intrinsic assignment; if it is a pointer component, the expression must be a valid target for it,[3] as in a pointer assignment statement (Section 3.12).

When 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
```

[3] In particular, it must not be a constant.

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)

```
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 31 letters[4] 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.

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
```

[4] An operator token may have up to 63 characters in Fortran 2003.

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	`==` `/=` `<` `<=` `>` `>=`
	`.eq.` `.ne.` `.lt.` `.le.` `.gt.` `.ge.`
Logical	`.not.`
Logical	`.and.`
Logical	`.or.`
Logical	`.eqv.` or `.neqv.`
–	dyadic (binary) defined operator

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 by adding the 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
```

and changing the interface block to

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

The result of 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.

3.9 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 3.8, we may write

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

(structure constructors were met in Section 3.8, too).

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 3.8) 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.

3.10 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 to 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 elements
             ! .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,

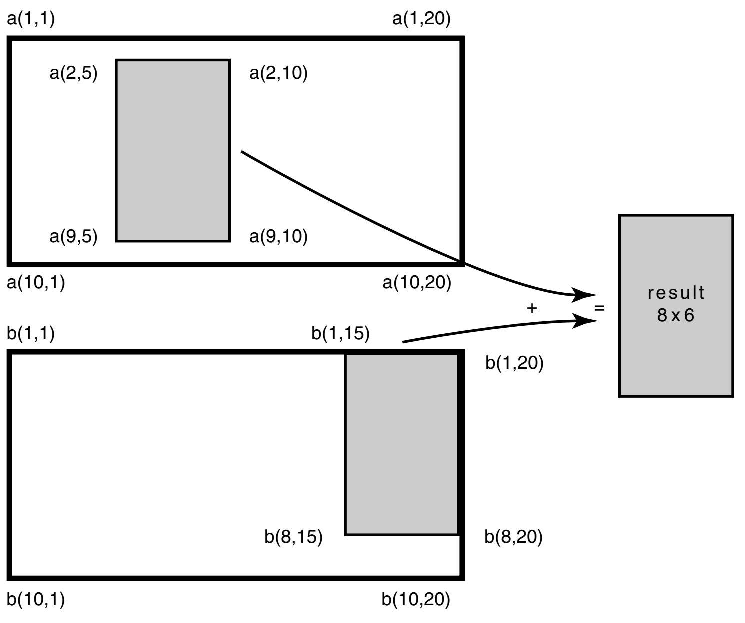
```
a(2:9,5:10) + b(1:8,15:20)
```

has element values

```
a(i+1,j+4) + b(i,j+14), i=1,2,...,8, j=1,2,...,6
```

This may be represented pictorially as in Figure 3.1.

Figure 3.1 The sum of two array sections.



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 6.11). 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 6.7.

3.11 Array assignment

By intrinsic assignment, an array expression may be assigned to an array variable of the same shape, 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)
```

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 8.3 and 8.4.

If a defined assignment (Section 3.9) is defined by an elemental subroutine (Section 6.11), 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 6.8 and assignment expressed with the help of indices in Section 6.9.

3.12 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 2.13, 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 8.15) 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 2.13 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 2.13 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 8.12.2) for the target, which means that an array section or array expression is always taken to have the value 1 for a lower bound and the extent for the corresponding upper bound.[5]

[5] In Fortran 2003, a lower bound may be specified, see Section 15.6

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.

3.13 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 8.2). For example, the end of the chain of Section 3.12 will be flagged as a disassociated pointer if the statement

```
nullify(first)
```

is executed initially to create a zero-length chain. Because often there are 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 6.5.3) should be executed instead.

3.14 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 .le. c             p .lt. t > 0
x-1 /= y             x+y < 3 .or. > 4.
d.lt.c.and.3.0       q.eq.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 3.8, the definitions of + given in Section 3.8, the definitions of assignment given in Section 3.9, 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)
```

This page intentionally left blank

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, that is a construct which begins with an initial keyword statement, may have intermediate keyword statements, and ends with a matching terminal statement, and that 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).

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

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*]

```
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 a `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 within one another 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 pre-defined 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

```
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 may be only 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.

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.

[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 7.4. In this section, all examples employ the simplest form of constant expression: the literal constant.

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
```

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 = expr1, expr2 [*`,`*expr3]*
  block
  `end do` *[name]*

where *variable* is a named scalar integer variable, *expr1, expr2,* and *expr3* (*expr3* is optional but 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 either both bear the same *name*, or both be unnamed.

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

`max((`*expr2*`-`*expr1*`+`*expr3*`)/`*expr3*`, 0)`

where `max` is a function which we shall meet in Section 8.3.2 and which returns either the value of the 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 the `do`, suppose we have used the type `entry` of Section 2.13 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.

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]Fortran 2008 allows a named `exit` to be used to exit from nearly any construct, not just a loop.

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
```

```
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 $1,2,\ldots$, *etc.* up to `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. From anywhere outside a `do` block, it is prohibited to jump into the block or to its `end do` statement.

It is similarly illegal 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 legal:

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

and

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

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, in Fortran 2008, is described in Section 20.4.1, and additional, but redundant, forms of do syntax in Appendix B.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 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 the most disputed statement in programming languages – the `go to` 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 which 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, creating *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` or `else` statement. It is permitted to pass control from a statment 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
```

4.6 Summary

In this chapter we have introduced the four main features by which the control in Fortran code may be programmed – the `go to` statement, the `if` statement and construct, the `case` construct, and the `do` construct. The effective use of these features is the 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 just 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 9, 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 4.7, which outputs the converted values of a temperature scale between specified limits, and Figure 4.8, which constructs a linked list. Valid inputs are shown at the end of each example.

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 coefficients 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

$$\begin{array}{c} 1 \\ 1\ \ 1 \\ 1\ \ 2\ \ 1 \\ 1\ \ 3\ \ 3\ \ 1 \\ 1\ \ 4\ \ 6\ \ 4\ \ 1 \\ \text{etc.} \end{array}$$

Write a program which reads an integer value `limit` and prints the coefficients of the first `limit` lines of this 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 2.13, 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.

Figure 4.7 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 4.8 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
```

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 the three kinds of program unit, the main program, external subprogram, and module.

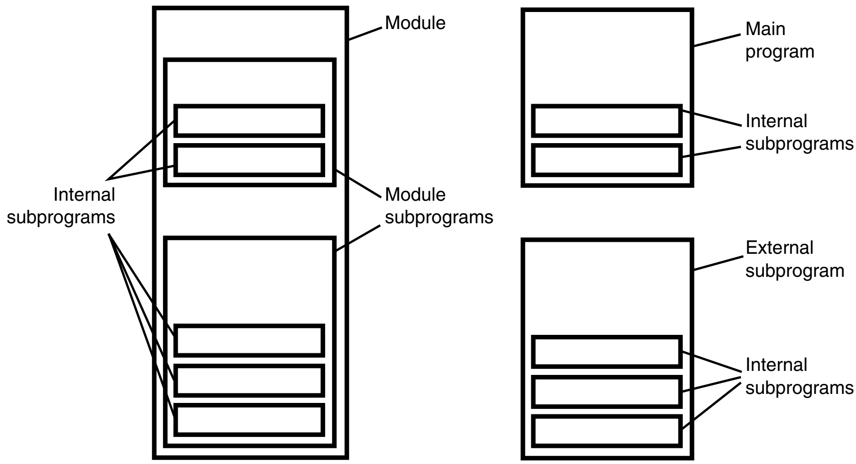
A complete program must, as a minimum, include one *main program*. This may contain statements of the kinds that we have met so far in examples, but normally its most important statements are invocations or *calls* to subsidiary programs known as *subprograms*. A subprogram defines a *function* or a *subroutine*. They 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 7.15). This collection may provide facilities associated with some particular task, such as providing matrix arithmetic, a library facility, or a data base. 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.

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-subprograms]*
> 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 complete program to stop. 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`, 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 flags the presence of one or more internal subprograms. We will describe internal subprograms in Section 5.6. They are excluded from the sequence of executable statements of the main program, which concludes with the last statement ahead of the `contains` statement followed by the `end` statement. The `end` statement may be the target of a branch from one of the executable statements. If the `end` statement is executed, the program stops.

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 character constant or a string of up to five digits whose leading zeros are not significant.[1] This might be used by a given processor to indicate the origin of the `stop` in a message. Examples are

```
stop
stop 'Incomplete data. Program terminated.'
stop 12345
```

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:

[1]Fortran 2008 allows any default integer or default character constant expression.

```
 subroutine-stmt
     [specification-stmts]
     [executable-stmts]
 [contains
       internal-subprograms]
 end [subroutine [subroutine-name]]
```

or

```
 function-stmt
     [specification-stmts]
     [executable-stmts]
 [contains
       internal-subprograms]
 end [function [function-name]]
```

The `contains` statement plays exactly the same role as within a main program (see Section 5.2). 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 7.15). 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-subprograms]
end [module [module-name]]
```

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 pre-defined 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 that might be required throughout a program (Section 2.6.2). 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 3.8, 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.

Figure 5.2 A module for interval arithmentic.

```
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
```

A module subprogram has exactly the same form as an external subprogram, except that `function` or `subroutine` *must* be present on the `end` statement. It always 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 will make it impossible for a module to access itself through other modules. It 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 7.6 and 7.10. For the present, we assume that the whole module is accessed without any renaming of the entities in it.

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. Note that `function` or `subroutine` must be present on the `end` statement. 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.

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 5.10 (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.

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

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.
```

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 contained in a `do` construct with this number of iterations, as shown in Figure 5.5.

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
```

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.[2] 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:

```
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
```

[2]The requirements on character length and shape agreement are relaxed in Appendix B.3.

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. 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 a 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 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
```

5.7.1 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 7.9).

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 (assumed-shape arrays are explained in Section 6.3)

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

5.7.2 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 6.3), 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), or 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 3.13) 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.

5.7.3 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 last executable statement in a main program is executed and does not cause a branch, the `end` statement is executed and the program stops. Similarly, if the last executable statement in a subprogram is executed and does not cause a branch, the `end` statement is executed and 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 7.5.4).

If a function specifies a defined operator (Section 3.8), the dummy arguments must have intent `in`. If a subroutine specifies defined assignment (Section 3.9), 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. It has been traditional for Fortran compilers not to check this rule, since they usually compile each program unit separately. Breaching the rule can lead to program errors at execution time that are very difficult to find. We recommend that all dummy arguments be given a declared intent. Not only is this good documentation, but it allows compilers to make more checks at compile time.

If a dummy argument has the `pointer` attribute, its `intent` is not allowed to be specified. This is because of the ambiguity of whether the intent applies to the target data object or to the pointer association.[3]

If a dummy argument is of a derived type with pointer components, its `intent` attribute also refers to the pointer association status of those components. For example, if the intent is `in`, no pointer assignment, allocation, or deallocation is permitted.

The Fortran 95 standard does not specify whether the `intent` attribute applies to the target of a pointer component.[4]

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 and the statement

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

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

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 thus:

```
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 7.9) from a previous invocation, or perform

[3] In Fortran 2003, `intent` is allowed and refers to the pointer association status (see Section 16.2).

[4] Fortran 2003 is clear that the `intent` attribute does not apply to the target of a pointer component.

Figure 5.6 A function that returns the distance between two points in space. The intrinsic function `sqrt` is defined in Section 8.4.

```
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
```

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 6.10.

A function result may be an array, in which case it must be declared as such.

A function result may also be a pointer.[5] 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 3.12), 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 6.5.3.

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 3.8. 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]However, it is not possible for a pointer to have a function as its target. In other words, *dynamic binding*, or association of a pointer with a function at run time, is not available. This deficiency is remedied in Fortran 2003 (see Section 13.6).

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 6.8) and `forall` statement (Section 6.9).

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 8) 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 7.2) 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 7.2) applied to the name, but this is not available in a module unless the function has the `private` attribute (see Section 7.6).

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;[6]
- 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
- a recursive procedure (Sections 5.16 and 5.17) or a pure procedure (Section 6.10) 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

[6] 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.

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 in the way described in the next section (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 3.8), as defined assignments (Section 3.9), 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.

5.12 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 5.7. 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 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.

On the user side, the code may look like that in Figure 5.8. 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 use of an interface block or, as a minimum, the procedure name would have to be declared in an `external` statement.

The procedure that is passed can only be an external or module procedure and its specific name must be passed when it also has a generic name (Section 5.18). Internal procedures are

Figure 5.7 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
```

not permitted[7] because it is anticipated that they may be implemented quite differently (for example, by in-line code), and because of the need to identify the depth of recursion when the host is recursive (Section 5.16) and the procedure involves host 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)
```

On 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 7.8). 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)
```

[7] A rule abolished in Fortran 2008, see Section 20.5.5.

Figure 5.8 Invoking the library code of Figure 5.7.

```
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
```

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 8.2). 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 5.9).

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, 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 5.9.

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.[8] Each is known as a *local* entity. This

[8] Apart from the effect of storage association, which is not discussed until Appendix B and whose use we strongly discourage.

Figure 5.9 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
```

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.[9]

A `use` statement of the form

`use` *module-name*

is regarded as a re-declaration 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 7.10 for a description of further facilities provided by the `use` statement when greater flexibility is required).

In the case of a derived-type definition, a module subprogram, or an internal subprogram, the name of an entity in the host (including an entity accessed by use association) is similarly treated as being automatically re-declared 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 use association. The host entity is said to be accessible by *host association*. For example, in the subroutine `inner` of Figure 5.10, `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.

[9] Apart from storage association effects (Appendix B).

Figure 5.10 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
```

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.[10] This allows an interface body to be constructed mechanically from the specification statements of an external procedure. Note, however, that if a derived type needed for the interface is accessed from a module, the interface block constructed from the procedure cannot be placed in the module that defines the type since a module is not permitted to access itself. For example, the attempted access in Figure 5.11 is not permitted.

Figure 5.11 Trying to write an interface in a module for a procedure that uses the module.

```
module m
   type t
      integer :: i, j, k
   end type t
   interface g
      subroutine s(a)
         use m        ! Illegal module access.
         type(t) :: a
      end subroutine s
   end interface
end module m
```

Within a scoping unit, each named data object, procedure, derived type, named construct, and namelist group (Section 7.15) 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

[10]In Fortran 2003, this is remedied by the `import` statement, Section 16.4.

any appearance of the name of an intrinsic procedure in another rôle makes the intrinsic procedure inaccessible by its name (the renaming facility described in Section 7.10 allows an intrinsic procedure to be accessed from a module and renamed). Within a 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 7.5.2) or an array constructor (Section 6.16) has a scope that is just the implied-`do`. It is different from any other entity with the same name.

5.16 Direct recursion

Normally, a subprogram may not invoke itself, either directly or indirectly, through a sequence of other invocations. However, if the leading statement is prefixed `recursive`, this is allowed. 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 5.12, which illustrates the use of a recursive function to sum the entries in a chain (see Section 2.13).

Figure 5.12 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
```

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 5.12). In fact, the result name, rather than the function name, must be used in any specification statement. In the 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 5.12, 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 7.2), but excepting those with the `data` or `save` attribute (see Sections 7.5 and 7.9) and any dummy arguments. The interface is explicit within the procedure.

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 5.13. For example, suppose that it is desired to integrate a function f of x and y

Figure 5.13 A 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
```

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 5.14. We can then integrate over x for a particular value of y, as shown in Figure 5.15, where `integrate` might be as shown in Figure 5.13. We may now integrate over the whole rectangle thus

```
volume = integrate(fy, ybounds)
```

Note that `integrate` calls `fy`, which in turn calls `integrate`.

Figure 5.14 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 5.15 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 name. Here, the interface block contains several interface bodies and the `interface` statement specifies the generic name. For example, the code in Figure 5.16 permits both the functions `sgamma` and `dgamma` to be invoked using the generic name `gamma`.

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 3.8).

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` above were defined in a module, the interface block becomes

Figure 5.16 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
```

```
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 3.8) or a defined assignment (Section 3.9) 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

generic-name
`operator(`*defined-operator*`)`

or

`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 5.18 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`.[11] 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* (have incompatible data type, kind, or rank) in order that any invocation will be unambiguous. The rule is that either

i) one of them has more non-optional data-object arguments of a particular data type, kind type parameter, and rank than the other has data-object arguments (including optional data-object arguments) of that data type, kind type parameter, and rank; or

ii) at least one of them has both

- 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 5.17 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.

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 5.7.1.

[11] Since intent must not be specified in Fortran 95 for a pointer dummy argument (Section 5.7.1), this implies that if an operand of derived data type also has the pointer attribute, it is the value of its target that is passed to the function defining the operator, and not the pointer itself. The pointer status is inaccessible within the function. In Fortran 2003, `intent` may be specified for a pointer dummy argument.

Figure 5.17 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
```

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 5.18 and 5.19, 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 8.6.1) may be invoked, as in Figure 5.20.

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

Figure 5.18 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 5.19 Use of the module of Figure 5.18.

```
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 illegal
   t = t + time(1.0)
   l = sqrt(s2)
   print *, v, t, l
end program test
```

Figure 5.20 A function with an argument of assumed character length.

```
   integer function count (letter, string)
      character (1), intent(in) :: letter
      character (*), intent(in) :: string
   !     Count the number of occurrences of letter in string
      count = 0
      do i = 1, len(string)
         if (string(i:i) == letter) count = count + 1
      end do
   end function count
```

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*`)` *]*

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 7.13; this, of course, must not be present on a `subroutine` statement.

Apart from `pure` and `elemental`, which will be explained in Sections 6.10 and 6.11, 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, type definitions, namelist groups, interface blocks, 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 8) 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 8.16.3) 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. Choosing a simple card game of your own choice, and using the random number procedure (Section 8.16.3), 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.

This page intentionally left blank

6. Array features

6.1 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 a notational convenience for the programmer, but also provide an opportunity to enhance optimization.

Arrays were introduced in Sections 2.10 to 2.13, their use in simple expressions and in assignments was explained in Sections 3.10 and 3.11, 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 8; the rich set of intrinsic procedures should be regarded as an integral part of the array features.

6.2 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 6.1 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 6.1 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).

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.

6.3 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 7.14 for the exact rules). If *lower-bound* is omitted, the default value is 1. 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` attribute is not regarded as an assumed-shape array because its shape is not necessarily assumed.

6.4 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 6.2.

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

Figure 6.2 A procedure with an automatic array. `size` is described in Section 8.12.2.

```
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
```

contains one or more values that are not known at compile time; that is, not a constant expression (Section 7.4). 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 7.14).

The other 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 6.3.

Figure 6.3 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.

Some small restrictions on the use of automatic data objects appear in Sections 7.5, 7.9, and 7.15.

[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.

6.5 Allocation of data

There is an underlying assumption in Fortran that the processor supplies a mechanism for managing heap[2] storage. The statements described in this section are the user interface to that mechanism.

6.5.1 The allocatable attribute

Sometimes an array is required to be of a size that is known only after some data have been read or some calculations performed. For this 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 of type integer
```

has been executed for it. Its allocation status is either *allocated* or *not currently allocated*. Its initial status is not currently allocated and it becomes allocated following successful execution of an `allocate` statement.

An important example is shown in Figure 6.4. 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 6.4 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))
   :
```

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

```
deallocate (a)
```

following which the array is 'not currently allocated'. The `deallocate` statement is described in more detail in Section 6.5.3.

If it is required to make any change to the bounds of an allocatable array, the array must be deallocated and then allocated afresh.[3] It is an error t allocate an allocatable array that is

[2]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.

[3]This restriction is removed in Fortran 2003, see Section 15.5.3.

already allocated, or to deallocate an allocatable array that is not currently allocated, but one that can easily be avoided by the use of the `allocated` intrinsic function (Section 8.12.1).

An undefined allocation status cannot occur. On return from a subprogram, an allocated allocatable array without the `save` attribute (Section 7.9) is automatically deallocated if it is local to the subprogram.[4] This automatic deallocation avoids inadvertent memory leakage.

6.5.2 The allocate statement

We mentioned in Section 2.13 that the `allocate` statement can also be used to give fresh storage for a pointer target directly. A pointer becomes associated (Section 3.3) following successful execution of the statement. The general form of the `allocate` statement is

`allocate (` *allocation-list* [`, stat=`*stat*] `)`

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 *stat* is a scalar integer variable that must not be part of an object being allocated.

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, program execution stops.

Each *allocate-object* is an allocatable array 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)))   ! illegal
```

or even

```
allocate (a(n), b(size(a)))         ! illegal
```

is not permitted, but

```
allocate (a(n))
allocate (b(size(a)))
```

[4] Strictly speaking, it is processor dependent as to whether an allocatable array remains allocated or is deallocated if it is local to a module and is accessed only by the subprogram, but such deallocation is not permitted in Fortran 2008 and we know of no Fortran 95 implementation that does it.

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 array, 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 4.8.

6.5.3 The deallocate statement

When an allocatable array 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*`] )`

where each *allocate-object* is an allocatable array 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.[5] 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). After an unsuccessful execution, each array that was not successfully deallocated retains its previous allocation or pointer association status. If `stat=` is absent and the deallocation is unsuccessful, program execution stops.

A pointer becomes disassociated (Section 3.3) following successful execution of the statement. 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.

A danger in using the `deallocate` statement is that storage may be deallocated while pointers are still associated with the targets it held. 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 (although, as noted at the end of Section 6.5.1, for allocatable arrays, there are circumstances in which this is automatic). 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 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.

[5] Note that this excludes a pointer that is associated with an allocatable array.

6.5.4 Allocatable dummy arguments

A dummy arrray is permitted to have the allocatable attribute. In this case, the corresponding actual argument must be an allocatable array 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 includes the bounds and may be 'not currently allocated'.

Our expectation is that some compilers will perform copy-in copy-out of the descriptor. Rule i) of Section 5.7.2 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 array itself, the situation is just like the case when the actual and dummy arguments are both explicit-shape arrays (see Section 5.7.3). Copy-in copy-out is permitted unless both arrays 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 array itself. If the intent is `in`, the array is not permitted to be allocated or deallocated and the value is not permitted to be altered. If the intent is `out` and the array 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 6.5.

Figure 6.5 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 (unit) n1, n2, n3
   allocate (array(n1, n2, n3))
   read (unit) array
end subroutine load
```

6.5.5 Allocatable functions

An array 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 6.6. The allocation status on each entry to the function is 'not currently allocated'. 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.

The interface must be explicit in any scoping unit in which the function is referenced. The result array is automatically deallocated after execution of the statement in which the reference occurs, even if it has the `target` attribute.

Figure 6.6 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
```

6.5.6 Allocatable components

Array components of 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.

Just as for an ordinary allocatable array, the initial state of an allocable component is 'not currently allocated'. This is also true for an ultimate allocatable component (Section 9.3) of an object created by an `allocate` statement. Hence, there is no need for default initialization of allocatable components. In fact, initialization in a derived-type definition (Section 7.11) of an allocatable component is not permitted.

In a structure constructor (Section 3.8), an expression corresponding to an allocatable component must be an array or a reference to the intrinsic function `null` with no arguments. If it is an allocatable array, the component takes the same allocation status and, if allocated, the same bounds and value. If it is an array, but not an allocatable array, 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 'not currently allocated'.

Allocatable components are illustrated in Figure 6.7, where code to manipulate polynomials with variable numbers of terms is shown.

Figure 6.7 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
```

Just as an allocatable array is not permitted to have the `parameter` attribute (be a constant), so an object of a type having an ultimate allocatable component is not permitted to have the `parameter` attribute; further, a structure constructor of such a type cannot be a constant and thus an constant expression cannot have such a type.[6]

When a variable of derived type is deallocated, any ultimate allocatable component that is currently allocated is also deallocated, as if by a `deallocate` statement. The variable may be a pointer or an allocatable array, 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 6.7) consists of the following steps for each such component.

i) If the component of *variable* is currently allocated, it is deallocated.

ii) If the component of *expr* is currently allocated, the component of *variable* is allocated with the same bounds and the value is then transferred using intrinsic assignment.

If the allocatable component of *expr* is 'not currently allocated', nothing happens in step ii), so the component of *variable* is left 'not currently allocated'. 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 5.7.2 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 6.5.5).

---

[6] All of these are permitted in Fortran 2003 provided the component is specifed as 'not currently allocated' explicitly with `null()` or implicitly by not being given a value. The component will always be 'not currently allocated'.

6.5.7 Allocatable arrays vs. pointers

Why are allocatable arrays needed? Is all their functionality not available (and more) with pointer arrays? 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 a pointer array 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 currently 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.

6.6 Elemental operations and assignments

We saw in Section 3.10 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 8) are elemental and have scalar dummy arguments that may be called with array actual arguments provided all the array arguments have the same shape. 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 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 function has an optional array-valued dummy argument that is absent, that dummy argument must not be used 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 3.11). 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 6.8 provides summation for scalars and rank-one arrays of intervals (Section 3.8). Alternatively, an elemental procedure can be defined for this purpose (Section 6.11).

Similarly, elemental versions of defined assignments may be provided explicitly or an elemental procedure can be defined for this purpose (Section 6.11).

6.7 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 6.8 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 6.9.

An array-valued function is not necessarily elemental. For example, at the end of Section 3.10 we considered the type

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

Figure 6.8 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
```

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 6.9 to provide matrix by vector multiplication.

6.8 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* 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 6.11).

Figure 6.9 A function for matrix by vector multiplication. `size` is defined in Section 8.12.

```
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
```

A single logical array expression may be used for a sequence of array assignments all of the same shape. The general form of this construct is

`where (`*logical-array-expr*`)`
   array-assignments
`end where`

The logical array 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 8.4) 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 8.11) 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. The masking expressions involved 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 that must be the final one. In addition, `where` constructs may be nested within one another; the masking expressions of the nested constructs must be of the same shape, as must be the array variables on the left-hand sides of the assignments.

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 6.10.

All the statements of a `where` construct are executed one by one in sequence, including the `where` and `elsewhere` statements. The logical array 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 6.10.

Figure 6.10 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
```

6.9 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)
:                       ! define names
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(index = lower: upper [:stride] &
          [, index = lower: upper [:stride]]... [,scalar-logical-expr] )
               [body]
        end forall [name]
```

where *index* is a named integer scalar variable. Its scope is that of the construct; that is, other variables may have the name but are separate and not accessible in the `forall`. The *index* may not be redefined within the construct. Within a nested construct, each *index* 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 6.13); they may not reference any *index* of the same statement but may reference an *index* 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* 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.[7]

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. *All such procedures must be pure* (see Section 6.10).

As in assignments to array sections (Section 6.13), 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.

[7]This is not actually a requirement, but any missing *index* 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.

6.10 Pure procedures

In the description of functions in Section 5.10, we noted the fact that, although it is permissible to write functions with side-effects, this is regarded as undesirable. In fact, used within `forall` statements or constructs (Section 6.9), the possibility that a function or subroutine reference might have side-effects is a severe impediment to optimization on a parallel processor – the order of execution of the assignments could affect the results. In order to control this situation, 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. In practical terms, this 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 7.9);

iv) performs no operation on an external file (Chapters 9 and 10); and

v) contains no `stop` statement.

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, and this intent must be `in` in the case of a function;

iii) any procedure internal to a pure procedure must be pure; and

iv) a variable that is accessed by host or use association or is an intent `in` dummy argument or any part of such a variable must not be the target of a pointer assignment statement; it must not be the right-hand side of an intrinsic assignment if the left-hand side is of derived type with a pointer component at any level of component selection; and it must not be associated as an actual argument with a dummy argument that is a pointer or has intent `out` or `inout`.

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 be used as such.

The main reason for allowing pure subroutines is to be able to use a defined assignment in a `forall` statement or construct and so, unlike pure functions, they may have dummy arguments that have intent `out` or `inout` or the pointer attribute. Their existence also gives the possibility of making subroutine calls from within pure functions.

All the intrinsic functions (Chapter 8) are pure, and can thus be referenced freely within pure procedures. Also, the elemental intrinsic subroutine `mvbits` (Section 8.8.3) is pure.

The `pure` attribute is given automatically to any procedure that has the `elemental` attribute (next section).

6.11 Elemental procedures

We have met already the notion of elemental intrinsic procedures (Section 6.6 and, later, Chapter 8) – 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 exists too 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 3.8 elemental, as shown in Figure 6.11. This is an aid to optimization on parallel processors.

Figure 6.11 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.
```

An elemental procedure must satisfy all the requirements of a pure procedure (previous section); in fact, it automatically has the `pure` attribute.[8] In addition, all dummy arguments and function results must be scalar variables without the pointer attribute. A dummy argument or its subobject may be used in a specification expression only as an argument to the intrinsic functions `bit_size`, `kind`, `len` or numeric inquiry functions of Section 8.7.2. An example is

```
elemental real function f(a)
   real, intent(in)                        :: a
   real(selected_real_kind(precision(a)*2)) :: work
   :
end function f
```

This restriction prevents character functions yielding an array result with elements of varying character lengths and permits implementations to create array-valued versions that employ ordinary arrays internally. A simple example that would break the rule is

[8] These requirements can be overridden in Fortran 2008 by the `impure` attribute, see Section 20.5.4.

```
elemental function c(n)
   character (len=n)   :: c       ! Invalid
   integer, intent(in) :: n
   real                :: work(n) ! Invalid
   :
end function c
```

If this were allowed, a rank-one version would need to hold `work` as a ragged-edge array of rank two.

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 6.2 (Section 6.4) perform its task on arrays of any shape or size, as shown in Figure 6.12. Calling `swap` with an array and a scalar argument is obviously erroneous and is not permitted.

Figure 6.12 Elemental version of the subroutine of Figure 6.2.

```
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 6.11) 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.

6.12 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 its 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.

6.13 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,

```
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 /)      ! Illegal
```

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.

When an array section with a vector subscript is an actual argument, 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 9.6).

In addition to the regular and irregular subscripting patterns just described, the intrinsic circular shift function `cshift` (Section 8.13.5) 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 6.12), 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 legal 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 6.12 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.

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   ! Illegal
```

for the array `tar` of Section 6.12. 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 6.12,

```
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 near the end of Section 6.5.2. 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`.

6.14 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 2.13. 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`.

6.15 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,[9] 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 8.12.2).

The facility provides a mechanism for subscripting or sectioning arrays such as

[9] In Fortran 2003, the `associate` statement provides a means of achieving this without the need for the `target` or `pointer` attribute, see Section 14.4.

```
tar%u
```

where `tar` is an array and `u` is a scalar component, discussed in Section 6.13. 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`.

6.16 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

(/ *array-constructor-value-list* /)

where each *array-constructor-value* is one of *expr* or *constructor-implied-do*. 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 = expr1, expr2* [, *expr3*])

where *variable* is a named integer scalar variable, and *expr1*, *expr2*, and *expr3* are scalar integer expressions. Its interpretation is as if the *array-constructor-value-list* had been written

max ((*expr2* - *expr1* + *expr3*)/*expr3*, 0)

times, with *variable* replaced by *expr1*, *expr1*+*expr3*,..., 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 6.7,

```
(/ (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*.

The type and type parameters of an array constructor are those of the first *expr*, and each *expr* must have the same type and type parameters. If every *expr*, *expr1*, *expr2*, and *expr3* is a constant expression (Section 7.4), the array constructor is a constant expression.

An array of rank greater than one may be constructed from an array constructor by using the intrinsic function `reshape` (Section 8.13.3). For example,

```
reshape( source = (/ 1,2,3,4,5,6 /), shape = (/ 2,3 /) )
```

has the value

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

Some deficiencies of array constructors have been removed in Fortran 2003 (details are in Section 15.9).

6.17 Mask arrays

Logical arrays are needed for masking in `where` statements and constructs (Section 6.8), and they play a similar role in many of the array intrinsic functions (Chapter 8). Often, such arrays are 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 8.11.1) 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.

6.18 Summary

We have explained that arrays may have zero size and that no special rules are needed for them. A dummy array may assume its shape from the corresponding actual argument. Storage for an array may be allocated automatically on entry to a procedure and automatically deallocated on return, or the allocation may be controlled in detail by the program. 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. 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 8.

We conclude this chapter with a complete program, Figures 6.13 and 6.14, 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 6.13 First part of a module for solving a set of linear equations. `size` is described in Section 8.12.2 and `maxloc` is described in Section 8.14.

```
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 6.14 Second part of Figure 6.13 module and a program that uses it. The edit descriptors used in the `write` statements are described in Section 9.12

```
!     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 6.12).

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 6.9 (Section 6.7) as a module procedure and supports the defined operations and assignments that it contains.

10. Using the type `stack` of Section 6.5.6, 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.

11. 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.

12. As 11., but with new bounds (1:6,1:9) and using the `reshape` intrinsic function.

This page intentionally left blank

7. Specification statements

7.1 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 a rudimentary `print` statement and with the `end` statement.

Already in Chapters 2 to 6, 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 Chapter 8.

7.2 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. This, and a prohibition on mixing types, is known as *strong typing*. In the case of Fortran, an entity that is not accessed by use or host association and is not explicitly typed by appearing in a type declaration statement is *implicitly* typed, being assigned a type according to the initial letter of its name.

The default in a program unit or an interface block 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.[1] This absence of strong 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 unforseen consequences. For this reason, we recommend that implicit typing be avoided. For no implicit typing whatsoever, the statement

```
implicit none
```

is available, and we recommend its use througout a whole program.

An `implicit none` statement may be preceded within a scoping unit only by `use` (and `format`) statements. An `implicit none` statement in a module applies to it, its module subprograms, and their internal subprograms. An `implicit none` statement in a main program or a subprogram, applies to it and its internal subprograms.

7.3 Declaring entities of differing shapes

So far, we have used separate type declaration statements such as

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

to declare several entities of the same type but differing shapes. In fact, 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)
```

7.4 Named constants and constant expressions

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

[1] See Section B.9 for means of specifying other mappings between the letters and types.

```
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, except within a complex constant (Section 2.6.3).[2] 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

```
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.

In this example, the expression `length**2` appeared in one of the array bound specifications. This is a particular example of a constant expression. Such an expression is expected to be evaluated at compile time, so it is restricted in its form.[3] A *constant expression* is an expression in which each operation is intrinsic, each exponentiation operator has an integer power and each primary is

i) a constant or a subobject of a constant;

ii) an array constructor whose expressions (including bounds and strides) have primaries that are constant expressions;

iii) a structure constructor whose components are constant expressions;

iv) an integer or character elemental intrinsic function reference whose arguments are constant expressions of type integer or character;

---

[2]This irregularity is corrected in Fortran 2003.

[3]These restrictions are relaxed in Fortran 2003, see Section 15.10.

v) a reference to one of the transformational intrinsic functions `repeat`, `reshape`, `selected_int_kind`, `selected_real_kind`, `transfer`, and `trim` with actual arguments that are constant expressions;

vi) a reference to the transformational intrinsic function `null` with an argument that is either of type other than character or has character length that is defined by a constant expression and is not assumed;

vii) a reference to

an array inquiry function (Section 8.12) other than `allocated`, the bit inquiry function `bit_size`, the character inquiry function `len`, the kind inquiry function `kind`, or a numeric inquiry function (Section 8.7.2)

where each argument is either a constant expression or a variable whose type parameters or bounds inquired about are neither assumed, defined by an expression other than a constant expression, defined by an `allocate` statement, nor defined by a pointer assignment;

viii) an implied-`do` variable with constant expressions as bounds and strides; or

ix) a constant expression enclosed in parentheses;

and where each subscript, section subscript, or substring bound 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.

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 named constant may be an array, as in the case

```
real, dimension(3), parameter :: field = (/ 0.0, 10.0, 20.0 /)
```

For an array of rank greater than one, the `reshape` function described in Section 8.13.3 must be applied.

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 array, dummy argument, or function result, since these are always variables. However, it may be of a derived type with a pointer component that is disassociated (Section 7.5.4):

```
type(entry), parameter :: e = entry(0.0, null())
```

Clearly, since such a pointer component is part of a constant, it is not permitted to be allocated or pointer assigned.

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
```

Finally, there is an important point concerning the definition of a scalar named constant of type `character`. Its length 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, there *is* a need to count when a character array is defined using an array constructor, since all the elements must be of the same length:[4]

```
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   '`.

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.

7.5 Initial values for variables

7.5.1 Initialization in type declaration statements

A 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 (Section 7.4), as in the examples

```
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 7.9). It must not be a dummy argument, a pointer, an allocatable array, an automatic object, or a function result.

[4] This restriction can be circumvented in Fortran 2003, see Section 15.9.

7.5.2 The data statement

An alternative way to specify an initial value for a 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 and structure constructors. A simple example is

```
real    :: a, b, c
integer :: i, j, k
data       a,b,c/1.,2.,3./, i,j,k/1,2,3/
```

in which a variable `a` acquires the initial value 1., `b` the value 2., etc.

If any part of a variable is initialized in this way, the variable automatically acquires the `save` atribute. The variable must not be a dummy argument, an allocatable array, an automatic object, or a function result. It may be a pointer and the corresponding value must be a reference to the intrinsic function `null` with no arguments.

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 named or literal constant:

```
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.

When the elements to be selected fall into a pattern which can be represented by `do`-loop indices, it is possible to write data statements a third way, like

```
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 6.16). A variable in an *expr* must be a *do-var* of an outer implied-do:

```
integer            :: j, k
integer, parameter :: l=5, l2=((l+1)/2)**2
real               :: a(l,l)
data ((a(j,k), k=1,j), j=1,l,2) / l2 * 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 7.1.

Figure 7.1 Result of an implied-do loop in a 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.

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 type definition of posn in Section 7.4, 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). Integer values may be binary, octal, or hexadecimal constants (Section 2.6.1).

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.

7.5.3 Pointer initialization and the function null

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 8.2). Pointers may be given the initial status of disassociated in a type declaration statement such as

```
real, pointer, dimension(:) :: vector => null()
```

or a `data` statement

```
real, pointer, dimension(:) :: vector
data vector/ 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 8.15), 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 Section 6.5.2, the structure constructor

```
entry (0.0, 0, null())
```

is available. Also, 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).

There is no mechanism to initialize a pointer as associated.[5]

[5] A restriction lifted in Fortran 2008.

7.5.4 Default initialization of components

Means are available 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 type definition (Section 2.9). If the component is not a pointer, this is done in the usual way (Section 7.5.1) 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.

Initialization does not have to apply to all components of a given derived type. An example for the type defined in Section 6.5.2 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.[6]

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 type definition such as

[6] However, an object of such a type that is declared in a module is required to have the `save` attribute unless it is a pointer or an allocatable array. This is because of the difficulty that some implementations would have with determining when a non-saved object would need to be re-initialized. It does not apply in Fortran 2008, where all data objects in a module have the `save` attribute.

```
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 7.5.2).

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 type definition is `private` or the components are `private`.

7.6 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 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. Thus, to make a generic procedure name accessible but the corresponding specific names inaccessible, we might write

```
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 (see Section 7.10). 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.

An object must not have the `public` attribute if its type was defined originally with the `private` attribute. Similarly, if a module procedure has a dummy argument or function result of such a type, the procedure must be given the attribute `private` and must not have a generic identifier that is `public`.[7]

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 7.11.

The `public` and `private` attributes may appear only in the specifications of a module.

[7]The restrictions of this paragraph have been lifted in Fortran 2003, see Section 16.14.

7.7 The pointer, target, and allocatable statements

For the sake of regularity in the language, there are statements for specifying the `pointer`, `target`, and `allocatable` attributes of entities. They take the forms:

```
pointer [::] object-name[(array-spec)]
                    [,object-name [(array-spec)]]...
target [::] object-name[(array-spec)]
                    [,object-name [(array-spec)]]...
```

and

```
allocatable [::] array-name[(array-spec)]
                         [,array-name [(array-spec)]]...
```

as in

```
real        :: a, son, y
allocatable :: a(:,:)
pointer     :: son
target      :: a, y(10)
```

We believe that it is much clearer to specify these attributes on the type declaration statements, and therefore do not use these forms.

7.8 The intent and optional statements

The `intent` attribute (Section 5.9) for a dummy argument that is not a dummy procedure or pointer may be specified in a type declaration 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 7.7, we believe that it is much clearer to specify these attributes on the type declaration statements, and therefore do not use these forms.

7.9 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 7.2. In this example, the local variables, `a` and `counter`, are initialized to zero, and it is assumed that their current values 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 7.2, 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 acquire the `save` attribute automatically (Section 7.5).

Figure 7.2 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 = 0.0
   else
      a = a + x
   end if
```

A similar situation arises with the use of variables in modules (Section 5.5). In theory, on return from a subprogram that accesses a variable whose scope is a module, the variable becomes undefined unless the main program accesses the module, another subprogram in execution accesses the module, or the variable has the `save` attribute. In practice, compilers treat module variables as having the `save` attribute.[8]

If a variable that becomes undefined 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 6.4). 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 recursive subprogram is shared by all instances of the subprogram.

An alternative to specifying the `save` attribute on a type declaration statement is the `save` statement:

`save` *[[::] variable-name-list]*

[8] In Fortran 2008, all data objects in a module have the `save` attribute.

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.

7.10 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 in the module named. Any `use` statements must precede other specification statements in a scoping unit. 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 (Section 7.15).

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-name `=>` *use-name*

and refers to a public entity in the module that is to be accessed by a different local name.

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 also for defined operators and assignments, where 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.

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-name =>] use-name

where each *access-id* is a public entity in the module, and is either a *use-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-name* or *access-id*. Where a *use-name* is preceded by a *local-name*, the entity is known locally by the *local-name*. 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-name* in an *only-list* being treated as the rename *use-name* => *use-name*). 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 7.3 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`.

An entity may be accessed by more than one local name. This is illustrated in Figure 7.4, where module `b` accesses `s` of module `a` by the local name `bs`; if a subprogram such as `c`

Figure 7.3 Making private an entity accessed from a module.

```
module one
   integer :: i
end module one
module two
   use one
   private
   :
end module two
```

accesses both `a` and `b`, it will access `s` by both its original name and by the name `bs`. Figure 7.4 also illustrates that an entity may be accessed by the same name by more than one route (see variable `t`).

Figure 7.4 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
```

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 7.2) and appears in no specification statements, then each appearance of the name will be taken, incorrectly, as a reference to the accessed variable. To avoid this, we 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.

7.11 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 B.2 and C.1.3) is

type *[[*, *access]*:: *]* *type-name*
  [private *]*
  component-def-stmt
  [component-def-stmt]...
end type *[type-name]*

Each *component-def-stmt* has the form

type [[,component-attr-list] ::]component-decl-list

where *type* specifies the type and type parameters (Section 7.13), each *component-attr* is allocatable, pointer, or dimension(*bounds-list*), and each *component-decl* is

*component-name [(bounds-list)][*char-len]*

or

*component-name [(bounds-list)][*char-len] [comp-int]*

The meaning of **char-len* is explained in Section 7.13 and *comp-int* represents component initialization, as explained in Section 7.5.4. 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 2.13), 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 6.14) 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 7.14) 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*.

Only if the host scoping unit is a module may the *access* qualifier or `private` statement appear. The *access* qualifier 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 for a type with `public` accessibility, the components of the type are inaccessible in any scoping unit accessing the host module, so that neither component selection nor structure construction are available there. Also, if any component is of a derived type that is `private`, the type being defined must be `private` or have `private` components.

We can thus distinguish three 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 `private` components, where the type is accessible but its components are hidden;

iii) all `private`, where both the type and its components are used only within the host module, and are hidden to an accessing procedure.

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. 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 B.2.1). 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.

7.12 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 [[, *attribute*]... ::] *entity-list* where *type* specifies the type and type parameters (Section 7.13), *attribute* is one of the following:

`parameter`	`dimension(`*bounds-list*`)`
`public`	`intent(`*inout*`)`
`private`	`optional`
`pointer`	`save`
`target`	`external`
`allocatable`	`intrinsic`

and each *entity* is

object-name [(*bounds-list*)] [**char-len*] [*=constant-expr*]

or

function-name [**char-len*]

or

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 7.13; a *bounds-list* specifies the rank and possibly bounds of array-valued entities.

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 *=constant-expr*; for example

```
real a, b, c(10)
```

If the statement specifies a `parameter` attribute, *=constant-expr* must appear.

If a `pointer` attribute is specified, the `target`, `intent`, `external`, and `intrinsic` attributes must not be specified. The `target` and `parameter` attributes may not be specified for the same entity, and the `pointer` and `allocatable` attributes may not be specified for the same array. If the `target` attribute is specified, neither the `external` nor the `intrinsic` attribute may also 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 `allocatable`, `parameter`, or `save` attribute must not be specified for a dummy argument or function result.

The `intent` and `optional` 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 8.1.3) 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.

7.13 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 )
```

in the `function` statement (Section 5.20), the component definition statement (Section 7.11), and the type declaration statement (Section 7.12). A *kind-value* must be a constant expression (Section 7.4) 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.

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 8.6.1) 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 not an asterisk must be a specification expression (Section 7.14). Negative values declare character entities to be of zero length.

In addition, it is possible to attach an alternative form of *len-value* to individual entities in a type declaration statement using the syntax *entity***char-len*, where *char-len* is either (*len-value*) or *len* and *len* is a scalar integer literal constant which specifies a length for the entity.

The constant *len* must 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.

7.14 Specification expressions

Non-constant scalar integer expressions may be used to specify the array bounds (examples in Section 6.4) and character lengths of data objects in a subprogram, and of function results. Such an expression may depend only on data values that are defined on entry to the subprogram. It must not depend on an optional argument, even if present. Any variable referenced must not 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.

Array constructors and derived-type constructors are permitted. The expression may reference an inquiry function for an array bound or for a type parameter of an entity which either is accessed by use or host association, or is specified earlier in the same specification sequence, but not later in the sequence.[9] An element of an array specified in the same specification sequence can be referenced only if the bounds of the array are specified earlier in the sequence.[10] Such an expression is called a *specification expression*.

An array whose bounds are declared using specification expressions is called an *explicit-shape array*.

A variety of possibilities are shown in Figure 7.5.

The bounds and character lengths are not affected by any redefinitions or undefinitions of variables in the expressions during execution of the procedure.

7.14.1 Specification functions

Any of the intrinsic functions defined by the standard may be used in a specification expression. In addition, a non-intrinsic pure function may be used provided that such a function is neither an internal function nor recursive, it does not have a dummy procedure argument, and the interface is explicit. Functions that fulfil these conditions are termed *specification functions*. The arguments of a specification function when used in a specification expression are subject to the same restrictions as those on specification expressions themselves, except that they do not necessarily have to be scalar.

[9] This avoids such a case as

```
character (len=len(a)) ::  fun
character (len=len(fun)) ::  a
```

[10] 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/)
```

Figure 7.5 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
```

As the interfaces of specification functions must be explicit yet they cannot be internal functions,[11] 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
```

[11]This prevents them enquiring, via host association, about objects being specified in the set of statements in which the specification function itself is referenced.

7.15 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 9.10. 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

The *namelist-group-name* is the name given to the group for subsequent use in the I/O statements. A variable named in the list must not be a dummy array with a non-constant bound, a variable with non-constant character length, an automatic object, an allocatable array, a pointer, or have a component at any depth of component selection that is a pointer, is allocatable or is inaccessible.[12] 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, type parameters, or shape 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.

[12] These restrictions are all lifted in Fortran 2003 except that assumed-size arrays remain prohibited.

7.16 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, 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 7.6, that uses a module to sort US-style addresses (name, street, town, and state with a numerical zip code) by order of zip code. It illustrates the interplay between many of the features described so far, but note that it is not a production code since the sort routine 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
```

Figure 7.6 A module to sort postal addresses and a program that uses it. `maxloc` is described in Section 8.14. The `read` and `write` statements here are explained in Sections 9.7 and 9.12.

```
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 significant 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. In the following module, identify all the scoping units and list the mappings for implicit typing for all the letters in all of them:

```
module mod
   implicit character(10, 2) (a-b)
   :
contains
   subroutine outer
      implicit none
      :
   contains
      subroutine inner(fun)
         implicit complex (z)
         interface
            function fun(x)
               implicit real (f, x)
               :
            end function fun
         end interface
      end subroutine inner
   end subroutine outer
end module mod
```

5. 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 2.13); or

b. write a type declaration statement for such a variable and a `data` statement to initialize its non-pointer components.

6. 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)`

This page intentionally left blank

8. Intrinsic procedures

8.1 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 a hundred intrinsic procedures in all, a particularly rich set. They fall into distinct groups, each of which we describe in turn. A list in alphabetical order, with one-line descriptions, is given in Appendix A.[1] 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.

8.1.1 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

[1] Appendix A also lists the few that were added to Fortran 2003 and the substantial number that were added to Fortran 2008.

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).

8.1.2 Categories of intrinsic procedures

There are four categories of intrinsic procedures.

i) *Elemental procedures* (Section 6.6).

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 6.10).

8.1.3 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.

8.1.4 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).

8.2 Inquiry functions for any type

The following are inquiry functions whose arguments may be of any type.

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 have the same type, type parameters, and rank as `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.

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`.

8.3 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.

8.3.1 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 is real and the 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 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 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 being the value of the argument `kind` if it is present or that of default complex otherwise. If `y` is absent, `x` may be of type integer, real, or complex. If `y` is present, it must be of type integer or real and `x` must be of type integer or real.

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; or
- complex, in which case the real part is truncated towards zero.

nint (a *[*,kind*]*) returns the integer value that is nearest to the real `a`. 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 default real when `a` is of type integer or real, and is that of `a` when `a` is type complex. The argument `a` may be of type integer, real, or complex. If it is complex, the imaginary part is ignored.

8.3.2 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 integer or real[2] values.

min (a1, a2 *[*,a3,...*]*) returns the minimum of two or more integer or real[2] values.

[2] In Fortran 2003, character type is also supported.

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. If `b` is zero, its sign is taken as positive. However, 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 8.7.1).

8.4 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 values x such that $|x| \leq 1$, expressed in radians in the range $0 \leq \mathrm{acos}(x) \leq \pi$.

asin (x) returns the arc sine (inverse sine) function value for real values x such that $|x| \leq 1$, expressed in radians in the range $-\frac{\pi}{2} \leq \mathrm{asin}(x) \leq \frac{\pi}{2}$.

atan (x) returns the arc tangent (inverse tangent) function value for real x, expressed in radians in the range $-\frac{\pi}{2} \leq \mathrm{atan}(x) \leq \frac{\pi}{2}$.

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 $-\pi < \mathrm{atan2}(y, x) \leq \pi$.[3] The values of x and y must not both be zero.

cos (x) returns the cosine function value for an argument of type real or complex that is treated as a value in radians.

cosh (x) returns the hyperbolic cosine function value for a real argument x.

exp (x) returns the exponential function value for a real or complex argument x.

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 $-\pi < w \leq \pi$.[4]

log10 (x) returns the common (base 10) logarithm of a real argument whose value must be positive.

[3] In Fortran 2003, if the arithmetic is IEEE, an approximation to $-\pi$ is returned if $x < 0$ and y is a negative zero.

[4] In Fortran 2003, 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.

sin (x) returns the sine function value for a real or complex argument that is treated as a value in radians.

sinh (x) returns the hyperbolic sine function value for a real argument.

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 of the result is not negative.[5]

tan (x) returns the tangent function value for a real argument that is treated as a value in radians.

tanh (x) returns the hyperbolic tangent function value for a real argument.

8.5 Elemental character and logical functions

8.5.1 Character–integer conversions

The following are elemental functions for conversions from a single character to an integer, and vice versa.

achar (i) is of type default 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 $0 \le i \le 127$, otherwise the result is processor dependent.

char (i *[*,kind*]*) is of type character and length one, with a 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 $0 \le i \le n-1$, 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)[6] is of type default integer and returns the position in the ASCII collating sequence of the default character `c`. If `c` is not in the sequence, the result is processor dependent.

ichar (c)[6] is of type default integer and returns the position of the character `c` in the processor collating sequence associated with the kind parameter of `c`.

[5] In Fortran 2003, 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.

[6] In Fortran 2003, there is an optional final `kind` argument that must be a scalar integer constant expression and controls the kind of the result.

8.5.2 Lexical comparison functions

The following elemental functions accept default character strings as arguments, 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.

8.5.3 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] )`[7] has type default integer and returns 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.

`len_trim (string)`[7] returns a default integer whose value is the length of `string` without trailing blank characters.

`scan (string, set [,back] )`[7] returns a default 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.

[7]In Fortran 2003, there is an optional final `kind` argument that must be a scalar integer constant expression and controls the kind of the result.

verify (string, set *[,back]*)[7] returns the default 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 left-most such character is returned. If `back` is present with value true, the position of the rightmost such character is returned.

8.5.4 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.

8.6 Non-elemental string-handling functions

8.6.1 String-handling inquiry function

len (string)[7] is an inquiry function that returns a scalar default 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.

8.6.2 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.

8.7 Numeric inquiry and manipulation functions

8.7.1 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 Binary Floating-point Arithmetic (IEEE 754-1985 or IEC 60559 : 1989) contains reals with $f_1 = 0$ (called denormalized 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 C.1.7). However, the function `sign` (Section 8.3.2) 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.

8.7.2 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 one 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.

8.7.3 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 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 $|\mathtt{x}\, b^{-e}|b^{p}$.

scale (x, i) returns a real with the same type parameter as `x`, whose value is $\mathtt{x}\, b^{\mathtt{i}}$, 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 $\mathtt{x}\, b^{\mathtt{i}-e}$.

spacing (x) returns a real with the same type parameter as `x` whose value is the absolute spacing of model numbers near `x`.

8.7.4 Transformational functions for kind values

There are two functions that return the least kind type parameter value that will meet a given numeric requirement. They have scalar arguments and results, so are classified as transformational.

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*]*) 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`, and decimal exponent range (as returned by the function `range`) at least `r`. 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). Both `p` and `r` are scalar integers; at least one of them must be present. If no corresponding kind value is available, the result is -1 if sufficient precision is unavailable, -2 if sufficient exponent range is unavailable, and -3 if both are unavailable.

8.8 Bit manipulation procedures

There are eleven procedures for manipulating bits held within integers. They are based on those in the US Military Standard MIL-STD 1753. They differ only in that here they are elemental, where appropriate, whereas the original procedures accepted only scalar arguments.

These intrinsics 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 8.7.1 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.

8.8.1 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`.

8.8.2 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 ≤ `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 ≤ `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` ≤ `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 ≤ `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.

`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 |`shift`| ≤ `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).

`not (i)` returns the logical complement of all the bits in `i`, according to the truth table

```
i              0   1
not(i)         1   0
```

8.8.3 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` ≤ `bit_size(from)`, `len` ≥ 0, `frompos` ≥ 0, `topos+len` ≤ `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`.

8.9 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.

8.10 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 8.11.1.

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, it 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.`'.

8.11 Transformational functions that reduce arrays

There are seven transformational functions that perform operations on arrays such as summing their elements.

8.11.1 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 described also in Section 6.17.

`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)`[8] returns the default integer value that is the number of elements of the logical array `mask` that have the value true.

`maxval (array)` returns the maximum value of an element of an integer or real[9] array. 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 or real[9] array. If `array` has size zero, it returns the largest positive value supported by the processor.

`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.

8.11.2 Optional argument dim

All these functions have an optional second argument `dim` that is a scalar integer. If this is present, 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))`.

As the rank of the result depends on whether `dim` is specified (unless the original is rank one), the corresponding actual argument must not itself be an optional dummy argument.

[8] In Fortran 2003, there is an optional final `kind` argument that must be a scalar integer constant expression and controls the kind of the result.

[9] In Fortran 2003, character type is also supported.

8.11.3 Optional argument mask

The functions `maxval`, `minval`, `product`, and `sum` have a third optional argument, a logical array `mask`. If this is present, it must have the same shape as 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.

8.12 Array inquiry functions

There are five functions for inquiries about the bounds, shape, size, and allocation status of an array of any type. Because the result depends on only the array properties, the value of the array need not be defined.

8.12.1 Allocation status

`allocated (array)` returns, when the allocatable array `array` is currently allocated, the value true; otherwise it returns the value false.

8.12.2 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]`* **`)`**[10] 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.

`shape (source)`[10] 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]`* **`)`**[10] 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]`*)[10] is similar to `lbound` except that it returns upper bounds.

[10]In Fortran 2003, there is an optional final `kind` argument that must be a scalar integer constant expression and controls the kind of the result.

8.13 Array construction and manipulation functions

There are eight functions that construct or manipulate arrays of any type.

8.13.1 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.

8.13.2 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.

8.13.3 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 $\mathtt{r}(s_1,\ldots,s_n)$ of the result, taken in subscript order for the array having elements $\mathtt{r}(s_{\mathtt{order}(1)},\ldots,s_{\mathtt{order}(n)})$, 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.

8.13.4 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 $\mathtt{source}(s_1,\ldots,s_n)$ where $(s_1,\ldots,s_n)$ is $(r_1,\ldots,r_{n+1})$ with subscript `dim` omitted (or `source` itself if it is scalar).

8.13.5 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.

8.13.6 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`. Element (*i, j*) of the result is `matrix(`*j*`,` *i*`)`.

8.14 Transformational functions for geometric location

There are two transformational functions that find the locations of the maximum and minimum values of an integer or real array.

`maxloc (array [, mask])`[11] returns a rank-one default 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 array `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 none, the result is processor dependent.[12]

`maxloc (array, dim [, mask])`[11] returns a default 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 ≤ `dim` ≤ 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 array `mask` when it is present. If there are none, the result is processor dependent.[12]

`minloc (array [, mask])`[11] is identical to `maxloc (array [, mask ])` except that the position of an element of minimum value is obtained.

`minloc (array, dim [,mask])`[11] is identical to `maxloc (array, dim [,mask])` except that positions of elements of minimum value are obtained.

8.15 Transformational function for pointer disassociation

The function `null` is available to give the disassociated status to pointer entities.

`null ([mold])` returns a disassociated pointer. The argument `mold` is a pointer of any type and may have any association status, including undefined. 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.

[11] In Fortran 2003, `array` may be of type character. There is also an optional final `kind` argument that must be a scalar integer constant expression and controls the kind of the result.

[12] In Fortran 2003, the result has all elements zero.

8.16 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.

8.16.1 Real-time clock

There are two subroutines that return information from the real-time clock, the first based on the ISO standard IS 8601 (Representation of dates and times). 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 of length 8 or more. Its first 8 characters are set to the century, year, month, and day in the form *ccyymmdd*.[13]

`time` is a scalar character variable of length 10 or more. Its first 10 characters are set to the time as hours, minutes, seconds, and milliseconds in the form *hhmmss.sss*.[13]

`zone` is a scalar character variable of length 5 or more. Its first 5 characters are set to 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`.[13]

`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 default integer[14] 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 default integer[14] holding the number of clock counts per second, or zero if there is no clock.

`count_max` is a scalar default integer[14] holding the maximum value that `count` may take, or zero if there is no clock.

[13] In Fortran 2003, the variable may be shorter; in this case the leftmost characters of the value are assigned to the variable. If the variable is longer than the value, the remaining characters are set to blank.

[14] In Fortran 2003, any kind of integer.

8.16.2 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.

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.

The start time is left imprecise because the purpose is to time sections of code, 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'
```

8.16.3 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.

8.17 Summary

In this chapter, we introduced the four categories of intrinsic procedures, explained the `intrinsic` statement, and gave detailed descriptions of all the procedures.

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 3.12 and 8.2, 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 3.12, 6.3, and 8.12.2, 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
```

This page intentionally left blank

9. Data transfer

9.1 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 *file* is used for a collection of data outside the main memory and a file is always organized into a sequence of *records*.

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 which 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.

9.2 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) be1d7dbf 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 later in this chapter; for the moment, it is sufficient to give a few examples. For instance, to print an integer value in a field of 10 characters width, we would use the edit descriptor `i10`, where `i` stands for integer conversion, and `10` specifies the width of the output field. To print a real quantity in a field of 10 characters, 5 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 10 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.

9.3 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 9.1, 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, provided they do not themselves cause further I/O to occur.[1]

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 9.1 could have been written

[1] This restriction was lifted in Fortran 2003 for internal files, and in Fortran 2008 for external files, see Sections 17.7 and 20.7.1.

Figure 9.1 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))
```

```
print '(3f10.3)', a(1:3)
```

However, no element of the array may appear more than once in an input item. Thus, the case in Figure 9.2 is not allowed.

Figure 9.2 An illegal input item (array element appears twice).

```
integer :: j(10), k(3)
:
k = (/ 1, 2, 1 /)
read '(3i10)', j(k)      ! Illegal because j(1) appears twice
```

If an allocatable array appears as an item, it must be currently allocated.

Any pointers in an I/O list must be associated with a target, and transfer takes place between the file and the targets.

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
```

Each ultimate component must be accessible (it may not, for example, be a `private` component of a `public` type).

It is convenient to extend the term ultimate component to include the case that ends with a component of derived type that is allocatable or is a pointer. For example, in Figure 9.3 the ultimate components of type `two` are `ordinary%comp`, `alloc`, and `point`, whereas `alloc%comp` and `point%comp` are not. The parent object (`obj` in the example) may be allocatable or a pointer.

Figure 9.3 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
type(two) :: obj
```

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. One reason for this restriction is because of the problems associated with recursive data structures. For example, supposing `chain` is a data object of the type `entry` of Figure 2.3 (in Section 2.13), it might hold a chain of length three, `chain%index`, `chain%next%index`, `chain%next%next%index` with `chain%next%next%next` a disassociated pointer. Another reason is that Fortran 2003 allows edit descriptors to be defined for data structures (see Section 17.2). Programmers can write procedures that are called as part of the I/O processing. Such a procedure is much better able to handle structures whose size and composition vary dynamically, the usual case for allocatable or pointer components.

An I/O list may include an implied-`do` list, as illustrated by the fifth `print` statement in Figure 9.1. 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.

9.4 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.

[2]Such an illegal association could be established by pointer association.

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 expression
                ! are concatenated.
```

or

```
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 `open` statement (Section 10.3) 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 9.9.

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 9.12.3).

9.5 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 disc, memory stick, and CD 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 CD from which we are reading the unit number 10, and to a hard disc 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 disc, 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 disc 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 disc 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 to these default units (usually 5 for input and 6 for output).[3]

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 the `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, the same as that used for `read` without a unit number.

A default character variable identifying an *internal file* (see next section).

9.6 Internal files

Internal files allow format conversion between various representations to be carried out by the program in a storage area defined within the program itself. There are two particularly useful applications, one to read data whose format is not properly known in advance, and the

[3] In Fortran 2003, these values may be accessed from an intrinsic module, see Section 16.5.

other to prepare output lists containing mixed character and numerical data, all of which has to be prepared in character form, perhaps to be displayed as a caption on a graphics display. The character data must be of default kind. The first application will now be described; the second will be dealt with in Section 9.8.

Imagine that we have to read a string of 30 digits, which might correspond to 30 one-digit integers, 15 two-digit integers, or 10 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 10 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 9.4), character constants are not delimited. A record may be read only if it is defined (which need not only be by an output statement). Records are padded with blanks, if necessary.

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 must be of default character type and non-default character items are not permitted in input/output lists. It may be used for list-directed I/O (Section 9.9), but not for `namelist` I/O (Section 9.10).[4]

[4] This restriction has been lifted in Fortran 2003.

9.7 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 [,list]
```

and with a unit it may take the form

```
read ([unit=]u, [fmt=]fmt [,iostat=ios]              &
          [, err=error-label] [,end=end-label]) [list]
```

where *u* and *fmt* are the unit and format specifiers described in Sections 9.4 and 9.5, `iostat=`, `err=`, and `end=` are optional specifiers which allow a user to specify how a `read` statement shall recover from various exceptional conditions, and *list* is a list of variables and implied-`do` lists of variables. 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 a part of a record or of several successive records.

The meanings of the optional specifiers are as follows.

- If the `iostat=` is specified, then *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 9.11), a different negative value if an endfile condition was detected on the input device (Section 10.2.3), 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 the `end=` is specified, then *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 the `err=` is specified, then *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 are not specified 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 9.4, in which `error` and `last_file` are subroutines to deal with the exceptions. They will normally be system dependent.

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 10.2.3); if there

Figure 9.4 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.
      :
```

is otherwise an error condition, the file position is indeterminate. An end-of-file condition occurs also 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 which 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.

9.8 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 [*`,`*list]*

and

`write (`*[*`unit=`*]u*`,` *[*`fmt=`*]fmt [*`,iostat=`*ios] [*`,err=`*error-label]* `)` *[list]*

where all the components have the same meanings as described for the `read` statement (Section 9.7). Note that the optional `fmt=` may be omitted only if the optional `unit=` is also omitted. An asterisk for *u* specifies the standard output unit, 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 9.5, 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 on a graphics display.

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

Figure 9.5 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'
```

```
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.

9.9 List-directed I/O

In Section 9.4, 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 the sake of carriage control (which is described in Appendix C.3), 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 Section 9.12. 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 which are allowed in a logical constant (those following `t` or `f`, see Section 9.12.2) 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

```
10b6.4b(1.,0.)b(2.,0.)btbtest/
```

(in which *b* stands for a blank, and blanks are used as separators), then `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.

9.10 Namelist I/O

It can be useful, especially for program testing, to input or output an annotated list of values. The values required are specified in a `namelist` group (Section 7.15), 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 a `nml=` specifier with that name.

When reading, only those objects which are specified in the input record and which do not have a null value become defined. All other list items remain in their existing state of definition or undefinition. It is possible to define the value of an array element or section without affecting the other portions of the array. When writing, all the items in the group are written to the file specified. This form of I/O is not available for internal files.[5]

The value for a scalar object or list of values for an array is preceded in the records by the name or designator and an equals sign which may optionally be preceded or followed by blanks. The form of the list of values and null values in the input and output records is as that for list-directed I/O (Section 9.9), except that character constants must *always* be delimited in input records and logical constants must not contain an equals sign. A `namelist` input statement terminates on the appearance of a slash in the list outside a character constant. 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

[5]But in Fortran 2003 it is available.

```
&food litres_of_milk=5, no_of_eggs=12 /
```

where we note that the order of the two values given is not the same as their order in the `namelist` group – the orders need not necessarily match. The value of `kilos_of_butter` remains unchanged. The first non-blank item in the record is an ampersand followed without an intervening blank by the group name. The slash is obligatory as a terminator. On output, a similar annotated list of values is produced, starting with the name of the group and ending with a slash. Here, the order is that of 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 /
```

On output, the names are always in upper case.

Where a subobject designator appears in an input record, all 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 the object is scalar and of intrinsic type, the equals sign must be followed by one value. If it is of derived type or is an array, the equals sign must be followed by a list of values of intrinsic type corresponding to the replacement of each derived-type value by its ultimate components and each array by its elements 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. This allows programmers to document the structure of a `namelist` input file line by line. The comment is in the usual format for comments. 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 may not occur in a character context.

9.11 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 a default 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 transfer 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 different negative value to 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 10.3 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.

9.12 Edit descriptors

In the description of the possible forms of a format specification in Section 9.4, 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, and only in the case where an input/output list is empty or specifies only zero-sized arrays may there be no edit descriptor at all in the format specification.

On a processor that supports upper- and lower-case letters,[6] 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.

9.12.1 Repeat counts

Edit descriptors fall into three classes: *data*, *control*, and *character-string*. The data edit descriptors may be preceded by a repeat count (a nonzero unsigned default integer literal constant), as in the example

```
10f12.3
```

Of the remaining edit descriptors, only the slash edit descriptor (Section 9.12.4) may have an associated repeat count. A repeat count may be applied to 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 components 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:

[6] Almost all systems support both cases these days and this is a requirement in Fortran 2008.

```
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 components 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 parenthesis itself if it has no repeat factor. This we call *reversion*.

9.12.2 Data edit descriptors

Values of all the intrinsic data types may be converted by the `g` edit descriptor. However, for reasons of clarity, it is described last. No form of value 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.[7]

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

[7] Additional forms allowed by Fortran 2003 appear in Section 17.5.

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 2.6.1). 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.

Real values may be converted by either `e`, `en`, `es`, or `f` edit descriptors. 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` 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 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 9.12.4) 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 9.12.4) 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`.

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 9.12.3 and 9.12.4, respectively).

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.

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 10 characters wide, and the non-default character variable of length 4 containing the value 国際標準 would appear in a field 4 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 10 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*.

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.

Any intrinsic data type values may be edited with the `g`*w.d* and `g`*w.d*`e`*e* (***general***) edit descriptor. When used for real or complex types, it is identical to the `e` edit descriptor except that an output value with magnitude *n* in the range

$$0.1 - 0.5 \times 10^{-d-1} \leq n < 10^{d} - 0.5$$

or zero when $d = 0$ is converted as if by an `f` edit descriptor, and followed by a number of blanks equal to the width of the exponent part as it would be specified by an `e` edit descriptor. The equivalent `f` edit descriptor is `f`w'`.`d', where $w' = w - 4$ for `g`w`.`d or $w - e - 2$ for `g`w`.`d`e`e, and $d' = d - k$ when n lies in the range

$$10^{k-1}(1 - 0.5 \times 10^{-d}) \leq n < 10^{k}(1 - 0.5 \times 10^{-d})$$

for $k = 0, 1, \ldots, d$ and $d' = d - 1$ when $n = 0$ and $d > 0$. This form is useful for printing values whose magnitudes are not well known in advance, and where an `f` conversion is preferred where possible, and an `e` 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).

Derived type values are edited by the appropriate sequence of edit descriptors corresponding to the intrinsic types of the ultimate components of the derived type.[8] An example is:

```
type string
   integer            :: length
   character(len=20)  :: word
```

[8] Fortran 2003 offers enhanced facilities for derived-type input/output (Section 17.2).

```
end type string
type(string) :: text
read (*, '(i2, a)') text
```

9.12.3 Character string edit descriptor

A *default character* literal constant without a specified kind parameter 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.

9.12.4 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 9.12.1). These latter descriptors we shall deal with first.

Control edit descriptors setting conditions

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). The default is given by the `blank=` specifier (Section 10.3) currently in effect for the unit or is null for an internal file. Whatever the default may then be for a file, it may be overridden during a given format conversion 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.

Leading signs are always written for negative numerical values on output. For positive quantities other than exponents, whether the signs are written depends on the processor. The `ss` (sign suppress) edit descriptor suppresses leading plus signs, that is the value 99 printed by `i5` is *bbb*`99` and 1.4 is printed by `e10.2` as *bb*`0.14E+01`. To switch on plus sign printing, the `sp` (sign print) edit descriptors may be used; the same numbers written by `sp,i5,e10.2` become *bb*`+99` and *b*`+0.14E+01`. The `s` edit descriptor restores the option to the processor. An `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. 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.

Scale factors apply to the input of real quantities under the `e`, `f`, `en`, `es`, and `g` edit descriptors, and are a means of scaling the input values. Their form is *k*`p`, where *k* is a default integer literal constant specifying the scale factor. The value 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.

The scale factor *k*`p` also affects output with `e`, `f`, or `g` editing, but has no effect with `en` or `es` editing. 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 with the exponent reduced by *k*, and the significand multiplied by 10^k. Thus 0.31×10^3, written after a `2p` edit descriptor under control of `e9.2`, will appear as `31.00E+01`. This gives a 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 (without a repeat count) may be omitted, but we do not recommend that practice 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.

Control edit descriptors for immediate processing

Tabulation in an input or output field can be achieved using the edit descriptors `t`*n*, `tr`*n* (and *n*`x`), 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,....)'
```

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.

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

```
bbb99
b
b
b
bb100
```

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.

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.

9.13 Unformatted I/O

The whole of 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 9.3) 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 10.1) 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.

9.14 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 in the next chapter) 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=`*i* specifier to the `read` and `write` statements, where *i* 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 9.6.

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 data base applications. A weakness is that the length of all the records must be the same,[9] though,

[9]This deficiency is avoided in Fortran 2003 with stream access, Section 17.6.

Figure 9.6 Write, read, and replace record 14. The `open` and `inquire` statements are explained in Sections 10.3 and 10.5.

```
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 array B to direct-access file in record 14
   write (nunit, rec=14) b
   :
   read (nunit, rec=14) a  ! Read the array back into array a
   :
   do i = 1, len/2
      a(i) = i
   end do
   write (nunit, rec=14) a !  Replace modified record
```

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 direct-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 10.1) 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.

9.15 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 9.7 are not permitted. This prevents dependencies occurring within the item itself.

Figure 9.7 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.

Finally, a function reference must not appear in an expression anywhere in an I/O statement if it causes another I/O statement or a `stop` statement to be executed.

9.16 Summary

This chapter has begun the description of Fortran's extensive I/O facilities. It has covered the formatted I/O statements, and their associated format specifications, and then turned to unformatted I/O and direct-access files.

The syntax of the `read` and `write` statements has been introduced gradually. The full syntax is

`read (`*control-list*`)` *[input-list]*

and

`write (`*control-list*`)` *[output-list]*

where *control-list* contains one or more of the following:

`unit=` *u*	`err=` *error-label*
`fmt=` *fmt*	`end=` *end-label*
`nml=` *nml-name*	`advance=` *advance*
`rec=` *i*	`size=` *size*
`iostat=` *ios*	`eor=` *eor-label*

A *control-list* must include a unit specifier and must not include any specifier more than once. The `iostat` and `size` variables 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 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 statements to output the state of a game of tic-tac-toe (noughts and crosses) to a unit designated by the variable `unit`.

3. 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.

4. Write separate list-directed input statements to fill each of the arrays of Exercise 1. For each statement write a sample first input record.

5. 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.

10. Operations on external files

10.1 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 disc, 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.

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 one or the other.

- A file never contains both formatted and unformatted records.

Finally, we note that no statement described in this chapter applies to internal files.

10.2 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.

10.2.1 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 10.2.3), 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 9.9 and 9.10). 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.

10.2.2 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.

10.2.3 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 normally 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 10.4), by an `open` statement for the same unit (Section 10.3), 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.

10.2.4 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. 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.

10.3 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.

`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.

`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.

`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 10.4). If `unknown` is specified, the status of the file is system dependent. This is the default value of the specifier, if it is omitted.

`access=` *acc*, where *acc* is a character expression that provides one of the values `sequential` or `direct`. 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.

`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-access connection.

`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. 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 (Section 10.5) 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.

`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 9.12.4, Embedded blanks). 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`.

`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.

`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.

`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.

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.

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=, delim=, pad=, 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.

10.4 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')
```

10.5 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 of default kind,[1] are as follows.

iostat= *ios* and **err=** *error-label*, have the meanings described for them in the `open` statement in Section 10.3. 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.

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.

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.

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 the `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.

access= *acc*, where *acc* is a character variable that is assigned one of the values `SEQUENTIAL` or `DIRECT` depending on the access method for a file that is connected, and `UNDEFINED` if there is no connection.

sequential= *seq* and **direct=** *dir*, where *seq* and *dir* are character variables that are assigned the value `YES`, `NO`, or `UNKNOWN`, depending on whether the file *may* be opened for sequential or direct access, respectively, or whether this cannot be determined.

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.

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, *rec* becomes undefined.

[1]Those of integer or logical type may be of any kind in Fortran 2003.

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 been yet 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.

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.

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).

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`.

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.

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.

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.

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`.

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`.

A variable that is a specifier in 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 9.6 in Section 9.14).

An example of the `inquire` statement, for the file opened as an example of the `open` statement in Section 10.3, 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      citiesbbbbb
acc      DIRECTbbbbb
seq      NObbbbbbbbb
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.

10.6 Summary

This chapter has completed the description of the input/output features begun in the previous chapter, 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.

11. Floating-point exception handling

11.1 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. The clear need for exception handling, something that had been left out of Fortran 95, led to a facility being developed on a 'fast track' as a Technical Report,[1] suitable for immediate implementation as an extension to Fortran 95. In this chapter, we describe the extensions to Fortran 95 that were detailed in this Report and which are all included in Fortran 2003. We also describe the few related Fortran 2003 features that were not in the Report, with a clear indication of this in each case.

Most computers nowadays have hardware based on the IEEE standard for binary floating-point arithmetic,[2] which later became an ISO standard.[3] 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.

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.

---

[1] Technical Report ISO/IEC TR 15580 : 1998(E).

[2] IEEE 754-1985, Standard for binary floating-point arithmetic.

[3] IEC 559 : 1989, Binary floating-point arithmetic for microprocessor systems.

11.2 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.

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 8.7.1),

$$x = 0$$

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* 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` properly rounds 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*.

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 $-\infty$, 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 $-\infty$ with the correct sign.

invalid occurs if the operation is invalid, for example, $\infty \times 0$, 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 11.9.3.

11.3 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 16.5.

The module is unusual in that all a code 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, the extent of support may be determined by the inquiry functions of Sections 11.8.2 and 11.9.2.

11.4 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 11.8.3). 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 11.8.3) or the subroutine `ieee_set_status` (Section 11.8.4). 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 11.8.2) 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.

11.5 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 11.8.3). Halting is not precise and may occur any time after the exception has occurred. The function `ieee_support_halting` (Section 11.8.2) 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.

11.6 The rounding mode

Some processors support alteration of the rounding mode during execution. In this case, the subroutine `ieee_set_rounding_mode` (Section 11.9.4) may be used to alter it. The function `ieee_support_rounding` (Section 11.9.2) 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.

Note that the value of a literal constant is not affected by the rounding mode.

11.7 The underflow mode (Fortran 2003 only)

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 11.9.4) may be used to alter it. The function `ieee_support_underflow_control` (Section 11.9.2) may be used to inquire whether this facility is available for a particular kind of reals.

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.

11.8 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 11.8.2. Whether control of halting is supported may be determined by the function `ieee_support_halting`, see Section 11.8.2. 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 11.3.

The module contains two derived types (Section 11.8.1), named constants of these types (Section 11.8.1), and a collection of generic procedures (Sections 11.8.2, 11.8.3, and 11.8.4). None of the procedures is permitted as an actual argument.

11.8.1 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.

11.8.2 Inquiry functions for IEEE exceptions

The module `ieee_exceptions` contains two inquiry 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 *[*,x*]*) returns `.true.` if the processor supports the exception `flag` for all reals (`x` absent) or for reals of the same kind type parameter as the real argument `x`. 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.`.

11.8.3 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.

11.8.4 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 as it was then.

Figure 11.1 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 11.1. There are no facilities for finding directly the value held within such a variable of a particular flag, rounding mode, or halting mode.

11.9 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 11.9.1), named constants of these types (Section 11.9.1), and a collection of generic procedures (Sections 11.9.2, 11.9.3, 11.9.4, and 11.9.5). None of the procedures is permitted as an actual argument.

11.9.1 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 (**Fortran 2003 only**)

```
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.

11.9.2 Inquiry functions for IEEE arithmetic

The module `ieee_arithmetic` contains the following inquiry 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.

ieee_support_datatype (*[x]*) returns .true. if the processor supports IEEE arithmetic for all reals (x absent) or for reals of the same kind type parameter as the real argument x. 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 ieee_copy_sign, ieee_scalb, ieee_logb, ieee_next_after, ieee_rem, and ieee_unordered must implement the corresponding IEEE functions.

ieee_support_denormal (*[x]*) returns .true. if the processor supports the IEEE denormalized numbers for all reals (x absent) or for reals of the same kind type parameter as the real argument x. Otherwise, it returns .false..

ieee_support_divide (*[x]*) returns .true. if the processor supports divide with the accuracy specified by the IEEE standard for all reals (x absent) or for reals of the same kind type parameter as the real argument x. Otherwise, it returns .false..

ieee_support_inf (*[x]*) returns .true. if the processor supports the IEEE infinity facility for all reals (x absent) or for reals of the same kind type parameter as the real argument x. Otherwise, it returns .false..

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 (x absent) or for reals of the same kind type parameter as the real argument x. Otherwise, it returns .false..

ieee_support_nan (*[x]*) returns .true. if the processor supports the IEEE Not-A-Number facility for all reals (x absent) or for reals of the same kind type parameter as the real argument x. Otherwise, it returns .false..

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 (x absent) or for reals of the same kind type parameter as the argument x. 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 (*[x]*) returns .true. if sqrt implements IEEE square root for all reals (x absent) or for reals of the same kind type parameter as the real argument x. Otherwise, it returns .false..

ieee_support_standard (*[x]*) returns .true. if the processor supports all the IEEE facilities defined in this chapter for all reals (x absent) or for reals of the same kind type parameter as the real argument x. Otherwise, it returns .false..

ieee_support_underflow_control (*[x]*) (**Fortran 2003 only**) returns .true. if the processor supports control of the underflow mode for all reals (x absent) or for reals of the same kind type parameter as the real argument x. Otherwise, it returns .false..

11.9.3 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 11.9.1.

`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 $|n - \mathtt{x/y}| = 1/2$, n is even. If the result value is zero, the sign is that of x.

ieee_rint (x, y) 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 $2^{\mathtt{i}}$x if this is within the range of normal numbers. If $2^{\mathtt{i}}$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 $2^{\mathtt{i}}$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.

11.9.4 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) (**Fortran 2003 only**) 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) (**Fortran 2003 only**) 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`.

The example in Figure 11.2 shows the use of these subroutines to store the rounding mode, perform a calculation with round to nearest, and restore the rounding mode.

Figure 11.2 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
```

11.9.5 Transformational function for kind value

The module `ieee_arithmetic` contains the following transformational function that is permitted in a constant expression (Section 7.4):

ieee_selected_real_kind (*[p] [, r]*) is similar to `selected_real_kind` (Section 8.7.4) except that the result is the kind value of a real `x` for which `ieee_support_datatype(x)` is true.

11.10 Examples

11.10.1 Dot product

Our first example, Figure 11.3, is of a module for the dot product of two real arrays of rank 1. 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.

Figure 11.3 Module for the dot product of two real rank-1 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
```

11.10.2 Calling alternative procedures

Suppose the function `fast_inv` is a 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 11.4, 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 11.4 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)
```

11.10.3 Calling alternative in-line code

This example, Figure 11.5, 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.

11.10.4 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

Figure 11.5 As for Figure 11.4 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
```

11.6 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.

11.10.5 Access to IEEE arithmetic values

The program in Figure 11.7 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 11.6 A reliable hypotenuse function.

```
real function hypot(x, y)

! In rare circumstances this may lead to the signaling of
! ieee_overflow.
! The caller must ensure that exceptions do not cause halting.
   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
```

Figure 11.7 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
```

This page intentionally left blank

12. Interoperability with C

12.1 Introduction

Fortran 2003 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 new syntax for defining sets of integer constants that is useful in this context.

12.2 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 12.1, 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 one. 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`.

They are all of type character with length one and kind `c_char` (or default kind if `c_char` has the value -1).

Table 12.1. 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`

12.3 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.

`c_loc (x)` is an inquiry function that returns a scalar of type `c_ptr` that holds the C address of its argument `x`, which must

i) have interoperable type and type parameters and be

   a) a variable that has the `target` attribute and is interoperable;
   b) an allocated allocatable variable that has the `target` attribute and is not an array of zero size; or
   c) an associated scalar pointer;

or

ii) be a non-polymorphic scalar, have no length type parameters, and be

   a) a non-allocatable, non-pointer variable that has the `target` attribute;
   b) an allocated allocatable variable that has the `target` attribute; or
   c) an associated pointer.

`c_funloc (x)` is an inquiry function that returns the C address of a procedure. The argument `x` is permitted to be a procedure that is interoperable (see Section 12.7) or a pointer associated with such a procedure.

`c_associated (c_ptr1` *[*`, c_ptr2`*]*`)` is an inquiry 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 with arguments

`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 with arguments

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 non-contiguous array section. 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 12.9.

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 12.10.

12.4 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 B.2.1), have type parameters, have the `extends` attribute (Section 14.2), or have any type-bound procedures (Section 14.6). 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 struct 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
```

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 struct type that contains a bit field, or a C struct type that contains a flexible array member.

12.5 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 (Appendix B.3).

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];
```

12.6 The value attribute

For the sake of interoperability, a new attribute, `value`, has been introduced for scalar 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. The only restriction on the type is that, if it is character, the character length must be known at compile time. The argument must not be a pointer, be allocatable, have intent `out` or `inout`, be a procedure, or have the `volatile` attribute (Section 16.3).

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

```
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
```

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.

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).

12.7 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 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. All the dummy arguments must be non-optional and interoperable. 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. 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.

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 entitities 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 12.1 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.

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.

Figure 12.1 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
```

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 11.8.4) 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.

12.8 Interoperability of global data

An interoperable module variable (or a common block, Appendix B.2.3, 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. 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 struct 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.2.2) 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.

12.9 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);
```

and the Fortran code is shown in Figure 12.2.

Figure 12.2 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)
 :
```

The rules on shape and character length disagreement (Appendix B.3) 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 12.3.

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.

Figure 12.3 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
```

12.10 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 12.4 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 struct declaration is:

```
struct pass {
  int lenc, lenf;
  float *c, *f;
};
```

the C function prototype is:

```
void simulation(struct pass *arrays);
```

and the C calling statement might be:

```
simulation(&arrays);
```

Figure 12.4 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 12.5 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.

12.11 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.

Figure 12.5 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) :: 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) :: problem
   type(problem_struct), pointer :: problem_ptr
   call c_f_pointer(problem, problem_ptr)
   deallocate(problem_ptr)
 end subroutine goodbye
end module lib_code
```

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.

13. Type parameters and procedure pointers

13.1 Introduction

This chapter combines the separate topics of type parameter extensions and procedure pointers.

The type parameter extensions consist of the addition of deferred type parameters, type parameter enquiry, and the ability to parameterize derived types.

The procedure pointer extension provides the ability to associate a pointer with a procedure, similar to the way dummy procedures become associated with actual procedures.

13.2 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 13.4.2.

Deferred type parameters can be given values by the `allocate` statement; see Section 15.4.1 for details. For allocatable variables, they can also be given values by assignment; see Section 15.5.2 for details.

13.3 Type parameter enquiry

The (current) value of a type parameter of a variable can be discovered by a *type parameter enquiry*. 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 enquiry 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 enquiry `ch%len` with the 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.

13.4 Parameterized derived types

Type parameters have been introduced for derived types, 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).

13.4.1 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 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 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. For example, see 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
```

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.

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
```

13.4.2 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 14.5).

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 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).

13.4.3 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 = kind('a')
   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`, nor does it conflict with the use of the intrinsic function `kind` within the type definition.

13.4.4 Derived type parameter enquiry

The value of a type parameter of a variable can be discovered by a type parameter enquiry, as with intrinsic types (see Section 13.3). For example, in

```
type(char_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.

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.

13.5 Abstract interfaces

In Fortran 95, 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). Furthermore, in Fortran 2003, there are several situations where this becomes impossible (procedure pointer components or abstract type-bound procedures).

For these reasons the *abstract interface* is introduced in Fortran 2003. 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.

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 a function), `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

`public`
`private`
`bind ( c [, name=`*character-string*`] )`
`intent (` *inout* `)`
`optional`
`pointer`
`save`

and a *proc-decl* is

procedure-name [=> *null-init*]

where *null-init* is a reference to the intrinsic function `null` with no arguments. (The `bind` attribute for procedures is described in Section 12.7.)

Each *proc-attr-spec* gives all the procedures declared in that statement the corresponding attribute. The initialization (to being a null pointer) may only appear if a procedure is a pointer.

13.6 Procedure pointers

A procedure pointer is a pointer that, instead of being associated with a data object, is associated with a procedure. 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.

13.6.1 Procedure pointer variables

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
```

13.6.2 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 13.2.

Figure 13.2 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 Section 13.6.3.

13.6.3 The pass attribute

When a procedure pointer component (or a type-bound procedure, Section 14.6) 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 13.3. The dummy argument to which the object is to be passed is known as the *passed-object dummy argument*.

Unless the type has the `sequence` attribute (Appendix B.2.1) or the `bind` attribute (Section 12.4), 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 13.3. Type extension is fully discussed in Chapter 14.

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 13.3).

Figure 13.3 Using the `pass` attribute to associate 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.

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 13.4.3).

3. 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.

14. Object-oriented programming

14.1 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 2003 by type extension, polymorphic variables, and type-bound procedures.

14.2 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 new type inherits all of the components of the old type and may have additional components. So an `employee` variable has the inherited components of `name`, `age`, and `id`, and the additional components of `number` and `salary`. Where the order matters, that is, in a structure constructor that does not use keywords[1] and in default derived type input/output (Chapter 9), the inherited components come first in their order, followed by the new components in their order.

[1]The use of keywords in structure constructors is new in Fortran 2003 and is described in Section 15.3.

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 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 14.6);

- to create a type with different effects for existing operations, by overriding 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` attribute (see Appendix B.2.1) or the `bind` attribute (see Section 12.4). An extended type must not be given the `sequence` or `bind` attribute.

14.2.1 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`.

14.3 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 such objects.[2] A polymorphic pointer may only be pointer-associated with a type-compatible target, a polymorphic allocatable variable may only be allocated to have a type-compatible allocation (see Section 15.4), and a polymorphic dummy argument may only be argument-associated with a type-compatible actual argument. Furthermore, if a polymorphic dummy argument is allocatable

[2] A non-polymorphic object is type-compatible only with objects of the same declared type.

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 14.6) 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 14.5).

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
```

14.3.1 Establishing the dynamic type

A polymorphic dummy variable only has its dynamic type established by argument association, which means that it does not vary during a single execution of the procedure, though it may be different on different invocations.

However, the dynamic type 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 15.5.3), the receiving variable takes on the dynamic type 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 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.

In Fortran 2008, the dynamic type of an allocatable variable can also change due to automatic reallocation, see Section 20.6.2.

14.3.2 Limitations on the use of a polymorphic variable

A polymorphic variable may appear in an input/output list only if it is processed by derived-type input/output (Section 17.2).

The variable in an intrinsic assignment statement is not permitted to be polymorphic (this is relaxed in Fortran 2008 for allocatable variables). However, if it is associated with a non-polymorphic variable, perhaps via the `type is` guard in a `select type` statement (see Section 14.5), assigning to the non-polymorphic variable will have the desired effect.

A polymorphic variable is not permitted to be an actual argument corresponding to an intent `out` assumed-size dummy argument (see Section B.3).

14.3.3 Polymorphic arrays and scalars

A polymorphic variable can be either an array or a scalar (including an allocatable scalar, see Section 15.5.1).

A polymorphic array is always homogeneous; that is, each array element has the same dynamic type. 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 circumlocution of using an array of derived type with a scalar polymorphic pointer or allocatable component can be used.

14.3.4 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
```

An unlimited polymorphic object cannot be referenced in any normal way; it can only be used as an actual argument, as the pointer or target in pointer assignment, or as the selector in a `select type` statement (see Section 14.5).

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 14.2.

When an unlimited polymorphic pointer is allocated, the required type and type parameter values must be specified in the `allocate` statement (Section 15.4).

14.3.5 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 at the beginning of this section (14.3). Where such an effect is required, type-bound procedures (Section 14.6.3) 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.[3]

14.4 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. 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 harder to understand 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, 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

[3] Fortran 2008 allows additional attributes to be used for generic resolution even in this case, see Section 20.5.7.

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 intrinsics `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.

14.5 The select type construct

To execute alternative code depending on the dynamic type 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 13.4.1. 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 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 each type guard and the `end select` statement.

The selector is a variable or an expression and the *associate-name* is associated with it within the block in exactly the same way as for an associate construct (previous section). 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 14.1 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.

Figure 14.1 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
```

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 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 14.2.

14.6 Type-bound procedures

Often, in object-oriented programming, one wishes to invoke a procedure to perform a task whose nature varies according to the dynamic type of a polymorphic object.

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 of the object.

They are called type-bound because the selection of the procedure depends on the type 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.

14.6.1 Specific type-bound procedures

The type-bound procedure section of a 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`.

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 14.3 objects of type `mytype` have

Figure 14.2 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
```

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`.

Figure 14.3 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
```

The full syntax of the statement declaring a specific type-bound procedure is

`procedure` *[* `(`*interface-name*`)` *] [[, binding-attr-list]* `::` *] tbp-name [=> proc-name]*

where each *binding-attr* is one of

`public` or `private`
`deferred`
`non_overridable`
`nopass` or `pass` *[* `(`*arg-name*`)` *]*

and *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 13.6.3. The `(`*interface-name*`)` appears if and only if the `deferred` attribute also appears; these are described in Section 14.7. An example of the case where it is not desired to pass the invoking object is shown in Figure 14.4.

If the `non_overridable` attribute appears, that type-bound procedure cannot be overridden during type extension (see Section 14.6.3). Note that `non_overridable` is incompatible with `deferred`, since that requires the type-bound procedure to be overridden.

Figure 14.4 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
```

14.6.2 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

`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.

For example, in Figure 14.5 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.

Figure 14.5 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
```

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, unnamed generics that are public are accessible wherever the type or an object of the type is accessible.

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 14.6 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
```

Figure 14.6 A generic type-bound operator.

```
module mycomplex_module
   type mycomplex
      private
      : ! data components not shown
   contains
      private
      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
      : ! many other operations and functions...
   end type
contains
   : ! procedures which implement the operations
end module
```

14.6.3 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 14.7, where the new type `charged_particle` inherits not only the components of `particle`, but also its type-bound procedures `momentum` and `energy`.

Figure 14.7 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
```

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 generic bindings) must satisfy the usual rules for generic disambiguation (Sections 5.18 and 14.3.5). A procedure that would be part of an inherited generic set may be overridden using its specific name.

For example, in Figure 14.8 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 14.8 Extending a type with overriding of type-bound procedures.

```
type mycomplex
   private
contains
   procedure          :: mycomplex_plus_mycomplex
   procedure          :: mycomplex_plus_real
   procedure, pass(b) :: real_plus_mycomplex
   generic            :: operator(+) => mycomplex_plus_mycomplex, &
                           mycomplex_plus_real, real_plus_mycomplex
end type mycomplex

type, extends(mycomplex) :: instrumented_mycomplex
   integer, public :: plus_operation_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
```

14.7 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.

Here is a simple example:

```
type, abstract :: file_handle
contains
   procedure (open_file), deferred, pass :: 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
```

Here, the intention is that extensions of the type would have components that hold data about the file and `open` would be overridden by a procedure that uses these data to open it.

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 14.9 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).

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.

14.8 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 B.2.1) or the `bind` attribute (Section 12.4).

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 '`*`').

Figure 14.9 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
```

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 from execution of a `return` statement), the final subroutine is invoked with the object as its actual argument. This also occurs 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 14.10. 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.

Termination of a program by an error condition, 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 14.11, 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 14.10 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
```

Figure 14.11 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
```

14.8.1 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 14.12, `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.

Figure 14.12 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)
```

14.9 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 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 routine to accept an extra argument, typically a `real` vector, and pass that to the user-defined function when it is called;

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 routine giving it extra information each time, until the quadrature routine 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 routine.

Figure 14.13 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
```

With type extension, the user can package up a procedure with any kind of required data, and the quadrature routine will pass the data through. Figure 14.13 shows the definition of the types concerned and an outline of the quadrature routine. Details not relevant to the function evaluation (such as the definition of the types for passing options to the routine, 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 14.14 shows how the user could do this for an arbitrary polynomial function.

Figure 14.14 Extending the Figure 14.13 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
```

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 routine as shown in Figure 14.15.

14.10 Type inquiry functions

Two new intrinsic functions have been added which compare 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 unallocatable allocatable variable, it will return false.

Figure 14.15 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)
```

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 associaton status.

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 14.3). 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.

This page intentionally left blank

15. Establishing and moving data

15.1 Introduction

Many relatively minor improvements have been made for manipulating data objects.

15.2 Mixed component accessibility

It is now possible for some components of a type to be `private` while others remain `public`. The `private` statement in a type, which previously set all components to be private, now merely sets the default accessibility of components to be private. The default accessibility for each component may be overridden or confirmed in the component definition statement, by specifying the `public` or `private` attributes. 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, the `debug_tag` field is public, exposing itself to the user 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 the value for that component is omitted.

15.3 Structure constructors

In Fortran 95, structure constructors look like function calls, except that keyword arguments are not allowed. In Fortran 2003, 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 15.1.

Figure 15.1

```
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
```

If a component of a type has default initialization, its value may be omitted in the structure constructor as if it were an optional argument.[1] 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.

If the derived type has type parameters, these are specified in parentheses immediately after the type name in its structure constructor. Again, if the type parameters have default values, they may be omitted, as in the example in Figure 15.2.

[1] Fortran 2008 also allows omission of values for allocatable components, see Section 20.1.4.

Figure 15.2

```
type character_with_max_length(maxlen, kind)
   integer, len    :: maxlen
   integer, kind   :: kind = kind('a')
   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')
```

15.4 The allocate statement

As well as determining array size, the `allocate` statement can now determine type parameter values, type (for a polymorphic variable), and value. This is controlled either by the inclusion of a type specification in the `allocate` statement:

`allocate (` *[type-spec* `::` *] allocation-list [* `,` `stat=`*stat]* `)`

where *type-spec* is the type name followed by the type parameter values in parentheses, if any, for both intrinsic and derived types; or by use of the `source=` clause for a single object:

`allocate (` *allocation [* `,` `source=`*source-expr] [* `,` `stat=`*stat]* `)`

where the *source-expr* is an expression with which the *allocation* is type-compatible (see Section 14.3). If *allocation* is for an array, *source-expr* may be an array of the same rank, otherwise *source-expr* must be scalar.

An `allocate` statement with a *type-spec* is *typed allocation*, and an `allocate` statement with a `source=` clause is *sourced allocation*. We now explain the new features.

15.4.1 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 15.3 must be specified as `*` because it is assumed.

Figure 15.3

```
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.

15.4.2 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`.

15.4.3 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 15.4.

Figure 15.4

```
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.

Because the bounds and shape of the allocated item are not taken from the source, making a clone of an array has to be done as follows:

```
class(t), allocatable :: a(:), b(:)
:
allocate (a(lbound(b,1):ubound(b,1)), source=b)
```

15.5 Allocatable entities

There are several extensions to the `allocatable` attribute in Fortran 2003, beyond those of the Technical Report and described in Chapters 2 to 10. We will now describe each of these in turn.

15.5.1 Allocatable scalars

The `allocatable` attribute (and hence the `allocated` function) may now 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 (unit) reclen
allocate (character(reclen) :: chdata)
read (unit) chdata
```

where `reclen` allows the length of `character` to be specified at run time.

Allocatable scalar components can also be used to construct data structures which do not leak memory (because they get deallocated automatically).

15.5.2 Assignment to an allocatable array

We explained in Section 6.5.6 that intrinsic assignment for an object containing allocatable components causes the automatic allocation or reallocation of any allocatable component that is not allocated and of the right shape. In Fortran 2003, for consistency with allocatable components, this automatic reallocation is extended to ordinary allocatable variables as well. This 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 variable `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.

15.5.3 Transferring an allocation

The intrinsic subroutine `move_alloc` has been introduced to move an allocation from one allocatable object to another.

`call move_alloc (from, to)` where:

> **`from`** is allocatable and of any type. It has intent `inout`.
>
> **`to`** is allocatable and of the same type and rank as `from`. It has intent `out`.
>
> After the call, the allocation status and target (if any) of `to` is that of `from` beforehand and `from` becomes deallocated.

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.
```

This 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.).

15.6 Pointer assignment

Two improvements have been made to the array pointer assignment statement. The first is that it is now 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 new facility for array pointer assignment is that the target of a multi-dimensional array pointer may be one-dimensional. 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 the lower bound. 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`.

15.7 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.

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 7.7).

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 15.5, 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 15.5

```
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
```

15.8 Renaming operators on the use statement

User-defined operators may now be renamed on the `use` statement, just as variable and procedure names may be. 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
```

15.9 Array constructor syntax

A well-recognized deficiency of array constructors in Fortran 95 is that they are somewhat inconvenient to use for `character` type; each element must have exactly the same character length. This is irritating to the user, who is thus required to pad character constants with blanks manually, to make them all the same length. For array constructors involving variables, this requirement is often not checkable at compile time, leading to potential run-time errors or strange results.

Another deficiency is that for zero-sized array constructors, it can be difficult if not impossible for the compiler to deduce the value of any length type parameters (in Fortran 95 this is limited to `character` type).

A less serious deficiency is that one cannot mix items of different type even when those items would be assignable to a common type (for example, having integer or real items in a complex array constructor).

Finally, when parenthesized expressions are array constructor items, and when array constructors are items inside parenthesized expressions and function references, it can be difficult to match the parentheses so that the array constructors end with `/)`.

All of these deficiencies have been addressed in Fortran 2003. To make it easier to match parentheses, an array constructor may be bracketed with square brackets, `[ ]`, instead of `(/ /)`.

To overcome the type deficiencies, an array constructor may now begin with an explicit specification of its type and type parameters. The syntax for an array constructor with a type specification is:

`(/` *type-spec* `::` *ac-value-list* `/)` or `[` *type-spec* `::` *ac-value-list* `]`

where the *type-spec* is the short form used in the `allocate` statement (Section 15.4). In this case, the array constructor values may have any type (and type parameters) that is 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
```

If *type-spec* is absent, the rules of Fortran 95 continue to apply: all items must have the same type and type parameters. This rule applies to parameterized derived types, too.

15.10 Specification and constant expressions

A specification expression (used for an array bound or length type parameter) may now reference a recursive function, so long as the function does not invoke the procedure containing that specification expression. It may contain a type parameter enquiry (Section 13.3) or a reference to an IEEE inquiry function (Section 11.9.2). Inside a derived-type definition, a specification expression may also reference any type parameter of the type being defined.

A constant expression is not as restricted as in Fortran 95. It may reference any elemental or transformational standard intrinsic function, or the function `ieee_selected_real_kind` of the intrinsic module `ieee_arithmetic`, as long as its arguments are all constant expressions. This includes the mathematical intrinsic functions (`sin`, `cos`, etc.). For example,

```
real :: root2 = sqrt(2.0)
```

is now a valid initialization. The exponentiation operator is not limited to an integer power.

All the inquiry functions may be referenced in a constant expression with the restrictions on their arguments that are given in item vii) of the list in Section 7.4. A type parameter enquiry (Section 13.3) may be used as long as the type parameter is not assumed, deferred, or defined by an expression other than a constant expression.

A constant expression may reference the `null` intrinsic function as long as it does not have an argument with a type parameter that is assumed or defined by an expression that is not a constant expression.

Finally, within a derived-type definition, a constant expression may reference a *kind* type parameter of the type being defined.

Exercises

1. 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.)

2. 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.

This page intentionally left blank

16. Miscellaneous enhancements

16.1 Introduction

This chapter collects together a number of miscellaneous enhancements made in Fortran 2003 that do not fit into any convenient category.

16.2 Pointer intent

The `intent` attribute has been extended to include pointers. For a pointer, the intent refers to the 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
```

16.3 The volatile attribute

The `volatile` attribute is a new attribute which may be applied only to variables. It 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`.

16.3.1 Volatile semantics

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 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. For example,

```
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
:
```

where `register_event_flag` is a routine, possibly written in another language, which ensures that `event_has_occurred` becomes true when the specified event occurs.

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 14.3), 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 16.1, 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 routines; in particular, `mpi_isend` which requires the `volatile` attribute on its first dummy argument.

Figure 16.1 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
```

16.3.2 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 16.3.3), 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 16.2, 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.

Figure 16.2 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
```

16.3.3 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 an actual argument persists through a procedure reference, as for example in the MPI call in Figure 16.1, this means that the procedure referenced must have an explicit interface and the corresponding dummy argument must be declared to be volatile.

If the dummy argument is volatile, 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 of assumed-shape and if the actual argument is an array pointer, the dummy argument must be a pointer or of 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 12.6) 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.

16.4 The import statement

One problem with procedure interface blocks in Fortran 95 is that an interface body does not access its environment by host association, and therefore cannot use named constants and derived types defined therein.

In particular, it is desirable in a module procedure to be able to describe a dummy procedure that uses types defined in the module. For example, in Figure 16.3, the interface body is invalid, because it has no access either to type `t` or to the constant `wp`.

Figure 16.3 An invalid interface block in a module procedure.

```
module m
   integer, parameter :: wp = kind(0.0d0)
   type t
      :
   end type t
contains
   subroutine apply(fun,...)
      interface
         type(t) function fun(f) ! Not allowed
            real(wp) :: f        ! Not allowed
         end function fun
      end interface
   end subroutine apply
end module m
```

This problem has been addressed 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. Figure 16.4 shows a correct interface body to replace the incorrect one in Figure 16.3.

Figure 16.4 The interface block of Figure 16.3 made valid by adding an `import` statement.

```
      interface
         function fun(f)
            import   :: t, wp
            type(t)  :: fun
            real(wp) :: f
         end function fun
      end interface
```

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. 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.

16.5 Intrinsic modules

Like an intrinsic function, an intrinsic module is one that is provided by the Fortran processor instead of by the user or a third party. A Fortran 2003 processor provides at least five intrinsic modules: `ieee_arithmetic`, `ieee_exceptions`, `ieee_features`, `iso_c_binding`, and `iso_fortran_env`, and may provide additional intrinsic modules.

Also like intrinsic procedures, 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).

The IEEE modules provide access to facilities from the IEEE arithmetic standard and are described in Chapter 11. The intrinsic module `iso_c_binding` provides support for interoperability with C and is described in Chapter 12.

The intrinsic module `iso_fortran_env` provides information about the Fortran environment, in the form of named constants as follows.

`character_storage_size` The size in bits of a character storage unit (only applicable to storage association contexts, see Appendix B.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.

`numeric_storage_size` The size in bits of a numeric storage unit (only applicable to storage association contexts, see Appendix B.2).

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`.

Intrinsic modules should always be used with an `only` clause, as vendors or future standards could make additions to the module.

16.6 Access to the computing environment

Intrinsic functions have been added to provide information about environment variables, and about the command by which the program was executed.

16.6.1 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.

16.6.2 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. 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.

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.

16.7 Support for internationalization

The internationalization capabilities have been improved by additional requirements on the processor's basic character set, decimal symbol control for floating-point input/output, and many improvements to the support for other character sets, in particular for the universal character set ISO/IEC 10646 (also known as Unicode).

16.7.1 Character sets

The Fortran character set now includes all the lower-case letters and many additional special characters from the ASCII character set. The additional special characters are

`~ ^ \ { } ` [ ] | # @`

Square brackets may be used to delimit array constructors (see Section 15.9); the others may appear only in comments, character literals, and character string edit descriptors.

The intrinsic function `achar` now takes an optional `kind` argument. This argument specifies the kind of the result. 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.

Similarly, the intrinsic function `iachar` now accepts a character of any kind, returning the ASCII code for that character if it is in the ASCII character set and a processor-dependent value otherwise.

The new intrinsic function `selected_char_kind` can be used to select a specific character set.

`selected_char_kind (name)` returns the kind value for the character set whose name is given by the character string `name`, or -1 if 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.

16.7.2 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.

16.7.3 ISO 10646 character set

ISO/IEC 10646 UCS-4 is a 4-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 7-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.

ISO 10646 character variables may be used as internal files; 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.

16.7.4 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 in Fortran 2003 by the `encoding=` specifier on the `open` statement. For example,

```
open (20, name='output.file', action='write', encoding='utf-8')
```

The `encoding=` specifier on the `inquire` statement 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 16.5 shows the I/O routines for a data processing application using these facilities.

Figure 16.5

```
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
```

16.7.5 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 *decimal point*, decimal points are used in input/output just as in Fortran 95.

While the mode is *decimal 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 decimal comma mode.

The `decimal=` clause may appear on the `open`, `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. If there is no `decimal=` clause on the `open` statement, the mode for the unit defaults to decimal point. The default for internal files is also decimal point. For the `read` and `write` statements, 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 decimal comma and decimal 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 decimal comma during list-directed or namelist input/output, a semicolon acts as a value separator instead of a comma.

16.8 Lengths of names and statements

The maximum length for names (Section 2.7) and operator tokens (Section 3.8) has been increased to 63 characters.

Statements were previously limited to 40 lines (20 lines in fixed form, see Appendix C.1.1); the maximum length in either form is now 256 lines. That is, up to 255 continuation lines are allowed.

One of the reasons for allowing longer statements is to handle source code that is automatically generated.

16.9 Binary, octal, and hexadecimal constants

Binary, octal, and hexadecimal ('boz') constants, previously only allowed in `data` statements, are now also allowed as a principal argument in a call of the intrinsic functions `cmplx`, `dble`, `int`, and `real` (not for an optional argument that specifies the kind).

For `int`, the 'boz' constant is treated as if it were an integer constant of the kind with the largest range supported by the processor. Thus,

```
integer :: i, j
data i/z'3f7'/
j = int(z'3f7')
```

gives both `i` and `j` the same value (in decimal, 1015).

For `dble` and `real`, it is treated as having the value that a variable of the same type and kind type parameter as the result would have if its internal representation were the bit pattern specified. This interpretation of the bit pattern is processor dependent. For `cmplx` with result of kind value `kind`, a 'boz' argument for either `x` or `y` provides the same value as `real(x,kind)` or `real(y,kind)`, so that it specifies the internal representation of one component of the result.

The advantage of allowing 'boz' constants in expressions only as arguments to these intrinsics is that there is no ambiguity in the way they are interpreted. There are vendor extensions that allow them directly in expressions, but the ways that values are interpreted differ.

16.10 Other changes to intrinsic procedures

The intrinsic functions `max`, `maxval`, `min`, and `minval` may now be used on values of type `character`.

If a set of array elements examined by `maxloc` or `minloc` is empty, the location of its maximum or minimum element is now deemed to have all subscripts zero (it was processor dependent in Fortran 95).

The following intrinsic functions now have an optional `kind` argument at the end of the argument list: `count`, `iachar`, `ichar`, `index`, `lbound`, `len`, `len_trim`, `maxloc`, `minloc`, `scan`, `shape`, `size`, `ubound`, and `verify`. This argument specifies the kind of integer result the function returns, in case 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`.

The `count`, `count_rate`, and `count_max` arguments of the intrinsic subroutine `system_clock` may now be of any kind of integer; this is to accommodate systems with a clock rate that is too high to be represented in a default integer. Additionally, the `count_rate` argument may now be of type `real` as well as `integer`; this is to accommodate systems whose clock does not tick an integral number of times each second.

The character arguments of `date_and_time` are now assigned their results and are not required to be long enough to hold the values.

Changes have been made to the intrinsic functions `atan2`, `log`, and `sqrt` for processors that distinguish between positive and negative real zero (on most computers, now that IEEE arithmetic is widespread). The intrinsic function `atan2(y, x)` now returns an approximation to $-\pi$ if `x`<0 and `y` is a negative zero since this is the limit as `y` $\rightarrow 0$ from below (previously it returned an approximation to π). For similar reasons, the intrinsic function `log(x)` now returns an approximation to $-\pi$ if `x` is of type complex with a real part that is less than zero and a negative zero imaginary part; and the intrinsic function `sqrt(x)` for complex `x` now returns a negative imaginary result if the real part of the result is zero and the imaginary part of `x` is less than zero.

16.11 Error message retrieval

The disadvantage of using the `stat=` clause on an `allocate` or `deallocate` statement is that it is impossible for the program to provide a sensible report of the error, because error codes are processor dependent.

To overcome this, the `errmsg=` clause has been added to these two statements. This takes a scalar default character string variable, and if an error condition occurs that is handled by `stat=`, an explanatory message is assigned to the `errmsg=` 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)
  :
end if
```

16.12 Enhanced complex constants

A complex constant may now 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 now permitted to appear in a constant expression, and provides all this functionality and more, there is very little use for this feature.

16.13 Interface block extensions

The `module procedure` statement (see Section 5.18) has been changed in Fortran 2003. The keyword `module` is now 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 can be used to avoid the Fortran 95 limitation that an external procedure could not appear in more than one interface block. For example, in

```
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
```

this allows the use of both the `*` and `.and.` operators for 'bitwise and' on values of type `bitstring`.

A generic name is permitted in Fortran 2003 to be the same as a type name. The generic name 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.

16.14 Public entities of private type

Entities of private type are no longer 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.

Exercises

1. 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 value is such that the 'cent' field is beyond the decimal precision, a string consisting of all asterisks should be returned.

2. Write a program that displays the sum of all the numbers on its command line.

17. Input/output enhancements

17.1 Introduction

In this chapter, we explain the enhancements to input/output processing that have been made in Fortran 2003. Non-default derived-type input/output (Section 17.2) allows the programmer to provide formatting specially tailored to a type and to transfer structures with pointer components. Asynchronous input/output (Sections 17.3 and 17.4) has been available as compiler extensions for many years and is now standardized. Since the advent of IEEE arithmetic many compilers have provided facilities for input/output of the exceptional values; this is now standardized (Section 17.5). Stream access (Section 17.6) allows great flexibility for both formatted and unformatted input/output. The remaining sections detail miscellaneous simple enhancements.

17.2 Non-default derived-type input/output

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, the `dt` edit descriptor specifies a character string and an integer array to control the action. An example is

```
dt 'linked-list' (10, -4, 2)
```

The character string may be omitted; this case is treated as if a string of length zero had been given. The parenthetical list of integers may be omitted, in which case an array of length zero is passed.

Such subroutines may be bound to the type as generic bindings (see Section 14.6.2) 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). 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 must be given a positive value. Otherwise, if an end-of-file or end-of-record condition occurs it must be given, respectively, the value `iostat_end` or `iostat_eor` of the intrinsic module `iso_fortran_env` (see Section 16.5). 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 before or after 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 9.14) or `pos=` (for stream access, Section 17.6) is not permitted in a child data transfer statement.

This feature is not available in combination with asynchronous input/output (Section 17.3).

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 17.1.

Figure 17.1 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 17.2 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
```

The module that implements this is shown in Figure 17.2. 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.

In the following example, Figure 17.3, 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 17.3 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
```

17.3 Asynchronous input/output

Input/output may be asynchronous, that is, 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`, a `close`, 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). The compiler is, of course, required to recognize all the new syntax.

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`) performs wait operations for all pending input/output operations for the file.

Asynchronous input/output is not permitted in conjunction with user-defined derived-type input/output (previous section) 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 12.6).

17.4 The asynchronous attribute

The `asynchronous` attribute for a variable has been introduced to warn 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 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 16.3), 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: 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 a pointer array, the dummy argument must be an assumed-shape or pointer array.

17.5 Input and output of IEEE exceptional values

Input and output of IEEE infinities and NaNs, previously done in a variety of ways as extensions of Fortran 95, is specified. All the edit descriptors for reals treat these 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.

17.6 Stream access input/output

Stream access is a new method of accessing an external file. It is established by specifying `access='stream'` on the `open` statement and may be formatted or unformatted.

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 4 bytes, the user could reasonably expect the first `write` to write exactly 4 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 statment 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)` 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.

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'
```

17.7 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 17.2); a child data transfer statement is recursive since it always executes while its parent is in execution. The only other situation in which execution of a recursive input/output statement is allowed, and this is an extension from Fortran 95, is for input/output to/from an internal file where the statement does not modify any internal file other than its own.[1]

17.8 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.

In combination with `advance='no'` or stream access (Section 17.6), 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.

17.9 Comma after a P edit descriptor

The comma after a P edit descriptor becomes optional when followed by a repeat specifier. For example, `1P2E12.4` is permitted (as it was in Fortran 66).

[1]Fortran 2008 allows additional cases, see Section 20.7.1.

17.10 The iomsg= specifier

Any input/output statement is permitted to have an `iomsg=` specifier. This 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.

17.11 The round= specifier

Rounding during formatted input/output 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`. It may be overridden by a `round=` specifier in a `read` or `write` statement with one of these values. The meanings are obvious except for the difference between `nearest` and `compatible`. Both refer to a closest representable value. If two are equidistant, which is taken is processor dependent for `nearest` and the value away from zero for `compatible`.

The rounding mode may also 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.

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 I/O rounding mode currently in effect behaves differently from the other rounding modes.

In Section 9.12.2, the formula for n in the `g` edit descriptor contains, twice, the value 0.5. This value is altered by some of the rounding modes, becoming 1 for `up`, and for `zero` if the value is positive; 0 for `down`, and for `zero` if the value is negative; and -0.5 for `nearest` if the lower value is even.

17.12 The sign= specifier

The `sign=` specifier has been added to the `open` statement. It can take the value `suppress`, `plus`, or `processor_defined` and controls the optional plus characters in formatted numeric output. It may be overridden by a `sign=` specifier in a `write` statement with one of these values. The mode may also be temporarily changed within a `write` statement by the `ss`, `sp`, and `s` edit descriptors, which are part of Fortran 95.

There is a corresponding specifier in the `inquire` statement that is assigned the value `PLUS`, `SUPPRESS`, `PROCESSOR_DEFINED`, or `UNDEFINED`, as appropriate.

17.13 Kind type parameters of integer and logical specifiers

The integer and logical specifiers that return a value (such as `nextrec=`) were limited to default kind in Fortran 95. Any kind is permitted in Fortran 2003.

17.14 More specifiers in read and write statements

The inquire statement specifiers `blank=` and `pad=` are now also available in the `read` statement, and `delim=` is available in the `write` statement.

17.15 Intrinsic functions for I/O status testing

Two new elemental intrinsic functions are provided for testing the I/O status value returned through the `iostat=` specifier. 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.

17.16 Some inquire statement enhancements

We have already met a number of new specifiers for the `inquire` statement (Section 10.5): `encoding=` (Section 16.7.4), `id=` and `pending=` (Section 17.3), `pos=` (Section 17.6), `iomsg=` (Section 17.10), `round=` (Section 17.11), and `sign=` (Section 17.12). Further, one existing specifier, `access=`, now has the additional possible value for *acc* of `STREAM` if the file is connected for stream access.

The following new, optional specifiers have not so far been described, and complete our description of the input/output enhancements.

`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`.

`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`.

`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.

`stream=` *stm*, where *stm* is a character variable that is assigned the value `YES` if `STREAM` is included in the set of allowed access methods for the file, `NO` if `STREAM` is not included in the set of allowed access methods for the file, and `UNKNOWN` if the processor is unable to determine whether or not `STREAM` is included in the set of allowed access methods for the file.

17.17 Namelist enhancements

Most of the restrictions on variables named in a `namelist` statement (Section 7.15) have been removed. The only one that remains is that an assumed-size array is not permitted.

Namelist I/O (see Section 9.10) is now available for internal files.

Exercises

1. 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.

2. 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 open with `sign='suppress'`?

This page intentionally left blank

18. Enhanced module facilities

18.1 Introduction

The module facilities of Fortran 95, while adequate for programs of modest size, have some shortcomings for very large programs. The extent of these shortcomings was not properly appreciated when the main features of Fortran 2003 were chosen and a straightforward solution was not devised until the development of Fortran 2003 was nearly complete. Therefore, instead of risking a delay to the whole of Fortran 2003, it was decided to define the submodule feature as an extension in a Technical Report,[1] with the promise that the next revision of Fortran would include it, apart from correcting any defects found in the field. Since the formal approval procedures for a Technical Report are simpler than those for a Standard, this was published in 2005, well ahead of the publication of Fortran 2008, expected in 2010. This is why this chapter appears here. The features are part of Fortran 2008.

The shortcomings of the module feature 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 way of modularization 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. (A method of avoiding this for some compilers is described in Appendix D.)

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.

The introduction of submodules gives other benefits, which we can explain more easily once we have described the feature.

[1] Technical Report ISO/IEC TR 19767: 2005(E).

18.2 Submodules

Submodules provide a way of structuring a module into component parts, which may be in separate files. All module procedures continue to have their interface defined in the module, but their implementation can be deferred to a submodule. 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.

18.2.1 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 18.1. 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.

Figure 18.1 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 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.

There is also a syntax that avoids the redeclaration altogether:

```
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
```

In this case, the whole interface is taken from the interface block, including whether it is a function or a subroutine and the name of the result variable if it is a function.

18.2.2 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.

A submodule is identified by the combination of the name of its ancestor module and the name of its parent, for example, `points:points_a` for the submodule of Figure 18.1. This allows two submodules to have the same name if they are descendants of different modules. This identifier is needed only to specify it as the parent in the `submodule` statement of a child, as in

```
submodule (points:points_a) points_b
```

18.2.3 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 for a submodule procedure with the `bind` attribute (see Section 12.7), such a procedure does not have a binding label and cannot have a `name=` specifier. This is because a user of a module might write a submodule with the module as its parent and containing a procedure that accesses private module entities by host association. However, without a binding label, there is no mechanism for the user to invoke such a 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.

18.2.4 Submodules and use association

A submodule is not permitted to access its ancestor module by use association; there is, after all, no need since it has access by host association. It may, however, access any other module by use association. In particular, it is possible for a submodule of module `a` to access module `b` and a submodule of module `b` to access module `a`. A simple example is where a procedure of module `a` calls a procedure of module `b` and a procedure of module `b` calls a procedure of module `a`. Because circular dependencies between modules are not permitted, without submodules this would require that `a` and `b` were the same module, or that a third module `c` be used (containing those parts which were mutually dependent).

18.3 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 (but see also Appendix D). 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.

19. Coarrays

19.1 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.

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. Objects with the new syntax 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 on another image. Bounds in square brackets in coarray declarations follow the rules of

[1]Although this is not required, it is anticipated that in early implementations each image will execute the same executable file on near-identical hardware.

assumed-size arrays since a coarray always exists on all the images. The upper bound for the last codimension is never specified, which 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 with some restrictions.

19.2 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`. 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 `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:*]
```

`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`.

[2] Delay while the image waits for data to be transferred to or from another image.

19.3 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,*]
```

Unless the coarray is allocatable (Section 19.7), the form for the codimensions in square brackets is the same as that for the dimensions in parentheses for an assumed-size array. The total number of subscripts plus cosubscripts is limited to 15.

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)` and `a(2:10)` are coarrays if `a` is the coarray 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 the coarray exists on all images). 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 19.4). 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`, shape `(/10,20/)`; its lower cobounds are `1, -1, 0` and its upper cobounds are `10, 8, 1`.

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
```

A coarray may be allocatable, see Section 19.7.

A coarray is not permitted to be a pointer, but a coarray may be of a derived type with pointer or allocatable components, see Section 19.8. Furthermore, because an object of type `c_ptr` or `c_funptr` has the essence of a pointer, a coarray is not permitted to be of either of these types. Although a coarray is permitted to have a component of `c_ptr` or `c_funptr`, these are nearly useless, see Appendix B.10.2.

19.4 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]
```

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 are not permitted). Subscripts or section subscripts must be used when the coarray has nonzero rank. For example, `a[2,3]` is not permitted as a shorthand for `a(:)[2,3]`.

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, since it refers to image `16`, but a reference to `z(:)[2,4]` is invalid, since it refers to image `17`. Like array subscripts, it is the programmer's responsibility to ensure that cosubscripts are within bounds 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 within a procedure reference, and this could be via a defined operation or defined assignment.

That an executing image is selected in square brackets has no bearing on whether the executing image evaluates the expression or assignment. For example, the statement

```
z[6] = 1
```

is executed by every image that encounters it, not just image 6. 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.[3] 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`

[3] Polymorphic coindexed objects are much more restricted, see Section 19.10.

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.

19.5 The sync all statement

Each image executes on its own without regard to the execution of other images except when it encounters special statements called *image control statements*. The programmer inserts image control statements to ensure that, whenever one image alters the value of a coarray variable or a variable with the `target` attribute, no other image still wants the old value, and that whenever an image accesses the value of a variable, it receives the wanted value – either the old value (before the update) or the new value (from the update). 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 image *P* before the barrier, the new value is available to all other images after the barrier. If an image references the value of a variable before the barrier, it obtains the value before crossing the barrier.

Figure 19.1 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 19.1 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.

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 corray is allocated or deallocated, see Section 19.7. Other image control statements are described in Sections 19.13 and a complete list is found in Section 19.13.7.

19.6 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, assumed shape, or allocatable, see Figure 19.2.

Figure 19.2 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 :: x(n, *)[*]              ! Assumed size
   real :: y(:, :)[*]              ! Assumed shape
   real, allocatable :: z(:)[:, :] ! Allocatable
   real, allocatable :: a[:]       ! Allocatable scalar
```

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 19.3.

The restrictions on coarray dummy arguments are:

- the actual argument must be a coarray (see Section 19.3 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 20.4.2), the actual argument must be *simply contiguous* (satisfies conditions given Section 20.4.3, 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 an allocatable 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

Figure 19.3 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
```

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 a component that is an allocatable coarray, it must not have intent `out`. This is because deallocating the coarray would require an implicit synchronization.

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 (see penultimate paragraph of this section).

A function result is not permitted to be a coarray or to have an ultimate component that is a coarray. Since functions are not invoked in lockstep on every image, it would not make sense to have a coarray result.

Allocatable coarrays may be declared in a procedure. They are discussed in Section 19.7.

The rules for resolving generic procedure references have not been extended to allow overloading of array and coarray versions because it would be ambiguous.

A pure or elemental procedure is not permitted to define a coindexed object or contain any image control statements (Section 19.13.7), 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.

An elemental procedure is not 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 component is required to have the `save` attribute. Note that in Fortran 2008, variables declared in the specification part of a module or submodule, as well as main-program variables, automatically have the `save` attribute. Again, this is because an unsaved non-allocatable coarray would be coming into existence on procedure invocation, requiring 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 a recursive procedure may need separate allocatable coarrays at more than one level of recursion.

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 teams and the images of one team may be calling the procedure with one coarray while the images of the other team are calling the procedure with another coarray or are executing different code.

Each image independently 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.

19.7 Allocatable coarrays

A coarray may be allocatable. The `allocate` statement is extended so that the cobounds can be specified, 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, the dynamic types must be the same on all images. Together, these restrictions ensure that the corrays exist on every image and are consistent.

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 `deallocate`, all images synchronize at the beginning of the same `deallocate` statement, and do not continue with the next statement until all images have finished the deallocation.

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 (see Section 20.5.3), 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 2003 allows the shapes or length parameters to disagree on the two sides of an intrinsic array assignment to an allocatable array (see Section 15.5.2); the system performs the appropriate reallocation. Such disagreement is not permitted for an allocatable coarray, since it would imply synchronization.

For the same reason, intrinsic assignment is not permitted to a polymorphic coarray.

19.8 Coarrays with allocatable or pointer components

A coarray is permitted to be of a derived type with allocatable or pointer components.

19.8.1 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 for 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 derived-type intrinsic assignments

```
z[q] = z ! The pointer component of z[q] may become undefined
z = z[q] ! The pointer component of z may become undefined
```

are executed on an image other than q. It can also happen in a procedure invocation 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, allocating one of those components 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)
```

In an intrinsic assignment to a coindexed object that is an allocatable array, 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.

19.8.2 Procedure pointer components

A coarray is permitted to be of a type that has a procedure pointer component or a type-bound procedure. A procedure reference through 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, a reference through a type-bound procedure (Section 14.6) is allowed provided the type is not polymorphic; this ensures that the type and hence the procedure is the same on all images.

19.9 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.[4].

If an object with an allocatable 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 an allocatable coarray component, no disagreement of allocation status or shape is permitted for the coarray component.

[4]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.

It is not permissible to add a coarray component by type extension unless the type already has one or more coarray components.

19.10 References to polymorphic subobjects

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 that has a polymorphic allocatable subcomponent.

19.11 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 by use association, host association, or in a `block` construct (see Section 20.5.3) 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. This is because the copy-in copy-out mechanism is forbidden when associating an asynchronous or volatile actual argument with an asynchronous or volatile dummy argument, but passing a coindexed object as an actual argument virtually requires copy-in copy-out to be done.

19.12 Interoperability

Coarrays are not interoperable, since C does not have the concept of a data object like a coarray. Interoperability of coarrays with UPC[5] might be considered in the future.

19.13 Synchronization

We have encountered barrier synchronization in Sections 19.5 and 19.7. 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.

19.13.1 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

[5] 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 19.1, 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, ... 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 19.1 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 (Appendix B.10.1). 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.

19.13.2 The sync images statement

For greater flexibility, the `sync images` statement

`sync images (` *image-set* `)`

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 array of rank one holding distinct image indices or an asterisk indicating all images except the invoking image.

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 19.4 shows an example that imposes the fixed order 1, 2, ... on images.

Figure 19.4 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 19.5, 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 19.5 Using `sync images` to make other images to 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
```

19.13.3 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* and *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 19.6 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 19.6 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.

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.

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 subcomponent of that type because this would create a new lock that might be locked initially.

19.13.4 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 new 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 section 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.

19.13.5 The sync memory statement and atomic subroutines

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 *atomic subroutines*, that are permitted to break the segment ordering rules of Section 19.13.1.

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 the use of the `sync memory` statement or atomic subroutines and defer their description to Appendix B.10.1.

19.13.6 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. They have the same role for these statements as they do for `allocate` and `deallocate` in Fortran 2003 (Section 16.11).

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` in the `iso_fortran_env` intrinsic module, and 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 19.14).

19.13.7 The image control statements

The full list of image control statements is

- `sync all` statement;
- `sync images` statement;
- `lock` or `unlock` statement;
- `sync memory` statement;
- `allocate` or `deallocate` statement involving a coarray;
- `critical` or `end critical` statement;
- `end` or `return` statement that involves an implicit deallocation of a coarray;
- a statement that completes the execution of a `block` (see Section 20.5.3) and results in an implicit deallocation of a coarray;
- `stop` or `end program` statement.

19.14 Program termination

It seems natural to allow all images to continue executing until they have all executed a `stop` or `end program` statement, provided none of them encounters an error condition that may be expected to terminate its execution. This is called *normal termination*. On the other hand, if such an error condition occurs on one image, 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 if it executes a `stop` or `end program` statement. All images synchronize execution at the second step so that no image starts the completion step until all images have finished the initiation step. The synchronization step allows its data to remain accessible to the other images until they all reach the synchronization step. Normal termination may also be initiated during execution of a procedure defined by a C companion processor.

An image initiates error termination if it executes a statement that would cause the termination of a single-image program but is not a `stop` or `end program` statement. 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]*

has been introduced, where *stop-code* is an integer or default character constant expression. When executed on one image, it initiates error termination there and hence causes 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 Sections 5.3 and 20.1.6.

The example in Figure 19.7 illustrates the use of `stop` and `error stop` in a climate model that uses two teams, one for the ocean and one for the 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 team 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 routines, 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.

19.15 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 the order of writes from images is important, synchronization and the `flush` statement are required, since the image is permitted to hold the data in a buffer and delay the transfers until either it executes a `flush` statement for the file or the file is closed.

Any other preconnected unit is connected on the executing image only, and the file is completely separate from any preconnected file on another image.

Figure 19.7 `stop` and `error stop` in a climate model.

```
program climate_model
   use, intrinsic :: iso_fortran_env, only: stat_stopped_image
   integer, allocatable :: ocean_team(:), atmosphere_team(:)
   integer :: i, sync_stat
   :
! Form two teams
   ocean_team = [ (i,i=1,num_images()/2) ]
   atmosphere_team = [ (i,i=1+num_images()/2,num_images()) ]
   :
! Perform independent calculations
   if (this_image() > num_images()/2) then
      call atmosphere(atmosphere_team)
   else
      call ocean(ocean_team)
   end if
! Wait for both teams 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 teams
   :
contains
   subroutine atmosphere (team)
      integer :: team(:)
      : ! 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 (team, stat=sync_stat))
      if (sync_stat == stat_stopped_image) then
         : ! Remaining atmosphere images preserve data in a file
         stop
      end if
   end subroutine atmosphere
```

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.

Although a file is not permitted to be connected to more than one image in Fortran 2008, it is expected that a forthcoming Technical Report will define such a facility.

19.16 Intrinsic procedures

The following intrinsic procedures are added. None are permitted in a constant expression. Again, we use italic square brackets *[ ]* to indicate optional arguments.

19.16.1 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.
>
> **`coarray`** is a coarray 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.

19.16.2 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`*[*`, dim`*]*`)` returns the set of cosubscripts of `coarray` that denotes data on the invoking image.

> **`coarray`** is a coarray of any type.
>
> **`dim`** is a scalar integer whose value is in the range $1 \leq$ `dim` $\leq n$ where n is the corank of `coarray`.
>
> If `dim` is absent, the result is 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. If `dim` is present, the result is a default integer scalar holding cosubscript `dim` of `coarray` for data on the invoking image.

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 team of images by adding an array argument holding the indices of the team and a scalar argument holding the position of the executing image in the team, assuming that the size of the team is a power of 2. In a main program, set up two teams and values in a coarray, then call your subroutine simultaneously for your two teams.

20. Other Fortran 2008 enhancements

Highlights of the other Fortran 2008 enhancements are a large number of new intrinsic functions, mostly mathematical special functions and bit manipulation, and features aimed at the high-performance market. The remaining features are all minor, mostly aimed at making it easier to write programs without providing significant new functionality.

20.1 Trivial syntactic conveniences

20.1.1 Implied-shape arrays

When defining a named constant that is an array, it is no longer 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' ]
```

In the case of a named array constant of higher dimension, the 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)`.

20.1.2 Implied-do loops in data statements

In Fortran standards up to Fortran 2003, expressions used for subscripts and implied-`do` bounds within an implied-`do` loop in a `data` statement were limited to combinations of constants, implied-`do` variables, and intrinsic operations. This was much stricter than the requirements for constant expressions elsewhere; for example, references to intrinsic functions were not allowed. These restrictions have now been relaxed, so that the requirements on these expressions are now identical to those on other constant expressions.

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/
```

is valid Fortran 2008; in previous standards the same effect could be achieved by

```
real :: a(10,7,3)
integer,parameter :: diagonal_size = min(size(a,1), size(a,2))
integer,parameter :: dim3_size = size(a, 3)
data ((a(i,i,j),i=1,diagonal_size),j=1,dim3_size)/21*1.0/
```

20.1.3 Type-bound procedures

A type-bound procedure declaration statement now takes a list of procedure bindings, so that multiple type-bound procedures can be declared in a single statement. For example, instead of

```
type mycomplex
   :
contains
   procedure :: i_plus_myc
   procedure :: myc_plus_i
   procedure :: myc_plus_myc => myc_plus
   procedure :: myc_plus_r
   procedure :: r_plus_myc
   :
end type
```

one may write

```
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.

20.1.4 Structure constructors

A structure constructor can omit the value for an allocatable component; this is equivalent to specifying `null()` for that component value. For example, given the type definition

```
type item
  character(:), allocatable :: name
  integer                   :: n_in_stock = 0
end type
```

the structure constructor `item()` is permitted, and is equivalent to `item(null())`. The omission of this feature from Fortran 2003 was really just an oversight.

20.1.5 Semicolons

A continuation line in the program is now permitted to begin with a semicolon. For example,

```
a = 1; b = 2; c = 3&
; d = 4; e = 5
```

This was invalid according to the Fortran 95 and 2003 standards, but the restriction was widely agreed to be a mistake and few compilers enforced it. It is now deemed to be acceptable by Fortran 2008.

20.1.6 The stop statement

The `stop` statement now accepts any default integer or default character scalar constant expression as the *stop code*, instead of only simple literals. For example,

```
character(*), parameter :: pu_name = 'load_data_type_1'
:
stop pu_name//': value out of range'
```

Furthermore, the value of an integer stop code is not limited to the range 0 – 99999, so the statement '`stop -2**20`' is valid.

Finally, the standard recommends that the stop code be written to the error file (unit `error_unit` of the intrinsic module `iso_fortran_env`) and, if it is an integer, that it be used as the *process exit status* if the operating system has such a concept (and 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.

The same recommendations apply to the `error stop` statement, that is, if it supplies an integer stop code, that that be used for the exit status, and that otherwise the exit status should be zero. This is somewhat at odds with typical operating system conventions where non-zero exit codes conventionally indicate error termination, especially since in other error-termination situations, such as an unhandled input/output error or allocation failure, the Fortran standard is silent on what the exit status should be. Again, it seems that this facility is difficult to use in a portable fashion.

For these reasons, we recommend the use of an informative message rather than an integer, for both the `stop` and `error stop` statements.

20.1.7 Exit from nearly any construct

The `exit` statement can now be used to transfer control to the end of an enclosing `associate`, `block`, `if`, `select case`, or `select type` construct. 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 20.1.

Note that an `exit` statement without a construct name still exits the innermost `do` construct. Since the different behaviours can easily confuse, we recommend that if this new `exit` (from

Figure 20.1 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:) ]
   end if adding_to_set
```

a non-`do` construct) is used in proximity to an `exit` from a `do` construct, both `exit` statements have construct labels.

The constructs that the new `exit` cannot be used for are the `critical` and `do concurrent` constructs (see Sections 19.13.4 and 20.4.1). It is also prohibited to exit an outer construct from within a `critical` or `do concurrent` construct.

20.2 Limitation changes

20.2.1 64-bit integer support

The maximum integer size is now required to have a range of at least 18 decimal digits; that is, the declaration

```
  integer(selected_int_kind(18)) :: big_int
```

will necessarily be accepted, and on a binary machine (all modern computers) this will be a 64-bit integer variable.

20.2.2 Maximum array rank

In accordance with the advent of 64-bit machines and the much larger memory sizes that are now available, the maximum rank of an array has been increased from 7 to 15. For example,

```
  integer, parameter :: ieee_single = ieee_selected_real_kind(6)
  real(ieee_single)  :: x(10, 10, 10, 10, 10, 10, 10, 10, 10, 10)
```

declares an array that has 10^{10} elements; since the size of a single-precision IEEE floating-point is four bytes, this requires 40 GB of memory.

If an array is also a coarray, the limit applies to the sum of the rank and corank.

20.3 Data expressiveness

20.3.1 Allocatable components of recursive type

An allocatable component is now 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 20.2 Allocatable list example.

```
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.
```

Figure 20.2 shows how this can be used to build a list. 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 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 20.3. Similar comments apply to element deletion, illustrated by subroutine `pop` in Figure 20.3.

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
```

Figure 20.3 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
```

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.)

20.3.2 Initial pointer association

The initial association status of a pointer can now be defined to be associated with a target, as long as that target has the `save` attribute and does not have the `allocatable` attribute. For example,

```
real, target  :: x(10,10) = 0
real, pointer :: p(:,:) => x
```

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.

20.4 Performance-oriented features

20.4.1 The do concurrent construct

A new 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 similar to that of the `forall` construct statement, including the new enhancements (see Section 20.5.6), namely

`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 affect 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):

- a `return` statement;
- an image control statement (see Chapter 19);
- a branch (for example, `go to` or `err=`) with a label that is outside the construct;
- a reference to a procedure that is not pure;
- 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`;
- an `exit` statement that would exit from the `do concurrent` construct; and
- a `cycle` statement that names an outer `do` construct.

By using `do concurrent` the programmer guarantees:

- 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 in only one iteration is not referenced or defined by any other iteration;
- any allocatable object that is allocated by more than one iteration is subsequently deallocated by those same iterations;
- any allocatable object that is allocated or deallocated by more than one iteration is not referenced, defined or deallocated by any iteration that does not allocate it first; and
- records (or positions) in a file are not both written by one iteration and read back by another iteration.

If records are written to a sequential file 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;
- documenting the parallelizability for code reading and maintenance; and
- as a crutch to compilers whose analysis capabilities are limited.

20.4.2 The contiguous attribute

The `contiguous` attribute is a new attribute for array pointers and assumed-shape dummy arrays. It specifies that the array will always be associated with a *contiguous* target or actual argument, and never be associated with a *non-contiguous* one. The basic idea is that a contiguous array is one where the elements are not separated by other data objects. An archetypal non-contiguous array is an array section with more than one element where adjacent elements in the section are not adjacent in the original (base) array; for example,

```
vector(::2)     ! all the odd-numbered elements
dtarray%re      ! the real parts of a complex array
```

In the first line, adjacent elements of the section are separated by one of the even-numbered elements of the original vector. In the second line (which uses the syntax of Section 20.5.1), adjacent elements of the section are separated by the imaginary components of the original array.

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 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 (a)` where `a` is an array of any type. This returns a default logical scalar with the value `.true.` if `a` is contiguous and `.false.` otherwise. If `a` is a pointer, it must be associated with a target.

Arrays in C are always contiguous, so `c_loc` was not permitted in Fortran 2003 for an array pointer or assumed-shape array. In Fortran 2008, `c_loc` is permitted for any target that is contiguous (at execution time). The example in Figure 20.4 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 new `c_sizeof` function to calculate the size of `x` in bytes (see Section 20.13.1).

Figure 20.4 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
```

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 runtime 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 amount 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 non-contiguous this will almost certainly perform much worse than if the `contiguous` attribute were not present.

For array pointers, the `contiguous` attribute has a runtime 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 non-contiguous 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.

20.4.3 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 which can been easily 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 15.6) – this previously permitted only rank-one arrays;
- 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 20.5 'flattens' the matrix `a` into a simple vector, and then uses that to associate another pointer with the diagonal of the matrix.

Figure 20.5 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 20.6, 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.

Figure 20.6 Contiguous buffer for asynchronous input/output.

```
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)
```

Another example of the use of simply contiguous to enforce contiguity of an actual argument is explained in Section 20.14.2.

Also, for argument association with a dummy coarray (see Section 19.6) 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 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 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 20.5.1);
  - 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.

20.5 Computational expressiveness

20.5.1 Accessing parts of complex variables

In Fortran 2003 and earlier, the real and imaginary parts of a complex variable were only accessible by the intrinsic functions `real` and `aimag`. This was inconvenient for updating a complex variable, so these can now be accessed as the pseudo-components `re` and `im` for the real and imaginary parts, respectively. For example,

```
complex :: impedance
impedance%re = 1.0
```

The `re` and `im` selectors can also be applied to complex arrays, where they 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
```

20.5.2 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. Fortran 2008 allows such a reference to a pointer function to be used in contexts that hitherto required 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 20.7.

20.5.3 The block construct

The `block` construct is a new kind of 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. 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
```

Figure 20.7 Example of accessor functions.

```
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)
```

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 name. 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.

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` statements 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. Statement function definitions, `common`, `equivalence`, and `namelist` statements are all prohibited because of potential ambiguities 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 20.1.7). Similarly, `block` constructs may be nested the same way that other constructs are nested. An example of this is shown in Figure 20.8.

Figure 20.8 Nesting `block` constructs.

```
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 2008, but various macro processors are widely used with Fortran.)

20.5.4 Impure elemental procedures

Elemental procedures as defined in Fortran 95 and 2003 are required to be *pure*, a condition which aids parallel evaluation. While this is advantageous for performance, it does prevent other possibilities where one wishes to perform *impure* processing elementally on arrays of arbitrary rank. In such cases, one was forced to provide a separate function for each permutation of conformant ranks; for a procedure with two arguments, that was 22 separate procedures (8 cases where both arguments had the same rank, 7 where the first was scalar and the second was an array, and 7 where the first was an an array and the second was scalar). With the increase of maximum rank to 15, this increases to $16+15+15=46$ separate procedures.

The `impure` prefix on the procedure heading allows one to define an impure elemental procedure, which processes array argument elements one by one in array element order. An example is shown in Figure 20.9. 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 the program with `stop` when too many errors have been encountered.

Figure 20.9 An impure elemental function.

```
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:

- all dummy arguments of an elemental procedure must be scalar non-coarray dummy data objects and must not have the `pointer` or `allocatable` attribute;
- all dummy arguments of an elemental procedure must have specified intent;
- the result variable of an elemental function must be scalar, must not have the `pointer` or `allocatable` attribute, and must not have a type parameter that is defined by an expression that is not a constant expression;
- in a reference to an elemental procedure, all actual arguments must be conformable; and
- in a reference to an elemental procedure, actual arguments corresponding to intent `out` and `inout` dummy arguments must either all be arrays or all be scalar.

20.5.5 Internal procedures as actual arguments

An internal procedure can be used as an actual argument or as the target of a procedure pointer. When it is invoked via the corresponding dummy argument or procedure pointer, it has access to the variables of the host procedure as if it had been invoked there. For example, in Figure 20.10, invocations of the function `fun` from `integrate` will use the values for the variables `freq` and `alpha` from the host procedure.

Figure 20.10 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 f(x)
    real(wp), intent(in) :: x
    f = x*sin(freq*x)/sqrt(1-alpha*x**2)
  end function
end subroutine
```

Apart from the convenience, this code can safely be part of a multi-threaded program because the data for the function evaluation is not being passed by global variables.

In the case of a procedure pointer associated with an internal procedure, 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 20.11, on return from `sub`, the variable `n` no longer exists for `f` to refer to.

20.5.6 Specifying the kind of a forall index variable

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 with `implicit none`, declaration of the `forall` index variable required the creation of a variable outside the `forall`.

Thus, Fortran 2008 allows the type and kind of the forall index to be specified in the `forall` statement itself; for example, in

Figure 20.11 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
```

```
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 it is completely hidden by the declaration in the `forall` statement.

20.5.7 Generic resolution

The set of specific procedures that are identified by the same generic identifier must satisfy stringent requirements to ensure that all possible references to the generic identifier can be unambiguously resolved to a specific procedure. For each pair of specific procedures, this usually requires that there be a dummy argument in one that is *distinguishable* from the corresponding dummy argument in the other (see Section 5.18). There are two extensions to what characteristics make a dummy argument distinguishable:

1) a dummy procedure is considered distinguishable from a dummy variable; and
2) a dummy argument with the `allocatable` attribute is considered distinguishable from a dummy argument with the `pointer` attribute when the pointer does not have intent `in`.

The interface block in Figure 20.12 illustrates the first extension: since the compiler always knows whether an actual argument is a procedure, no reference to `g1` could ever be ambiguous.

The interface block in Figure 20.13 illustrates the second extension: 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 20.12 Generic disambiguation based on procedureness.

```
interface g1
  subroutine s1(a)
    real a
  end subroutine
  subroutine s2(a)
    real, external :: a
  end subroutine
end interface
```

Figure 20.13 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 20.14.2) that would have made it possible to write an ambiguous reference.

20.6 Data usage and computation

20.6.1 Enhancements to the allocate statement

In Fortran 2003, to 'clone' an array using the `allocate` statement with the `source=` specifier, the bounds had to be specified on the allocation; for example

```
real, allocatable :: a(:), b(:)
:
allocate (b(lbound(a, 1):ubound(a, 2)), source=a)
```

In Fortran 2008 bounds may be omitted, in which case they will be taken from the `source=` specifier, allowing the much simpler

```
allocate (b, source=a)
```

Fortran 2008 also adds the 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, the restriction in Fortran 2003 that limits the `source=` specifier to acting on a single *allocation* has been lifted: both `mold=` and `source=` may be used when allocating multiple objects, for example,

```
allocate (a(10), b(20), source=173)
```

20.6.2 Automatic reallocation

Intrinsic assignment to an allocatable polymorphic variable is now allowed, and this extends the automatic reallocation feature introduced by Fortran 2003 for array shape and deferred type parameters to handle types.

If the variable is allocated and its dynamic type differs 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.

20.6.3 Elemental subprogram restrictions

In Fortran 2003, a dummy argument of an elemental subprogram was not permitted to be used in a specification expression for a local variable. The purpose of this restriction was to facilitate optimization of such procedures by ensuring that the space needed by local variables would be fixed for the whole array. However, the restriction was easily subverted by using allocatable variables or internal procedures, and proved not to be particularly useful in practice (it being trivial for the compiler to detect whether such a restriction held and therefore whether any relevant optimizations could be applied anyway). Therefore, this restriction has been removed in Fortran 2008.

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.

20.7 Input/output

20.7.1 Recursive input/output

A *recursive input/output* statement is an input/output statement that is executed as a result of a function reference in an I/O list. In Fortran 2003, recursive input/output was permitted but only for reading and writing from internal files. In Fortran 2008, 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
      :
```

20.7.2 The newunit= specifier

A longstanding inconvenience in Fortran programs has been the need to manually manage input/output unit numbers. This inconvenience 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.

These inconveniences are solved by the `newunit=` specifier on the `open` statement. 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 :: in
open (file='input.dat', status='old', form='unformatted', newunit=in)
:
read (in) data
```

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.

20.7.3 Writing comma-separated values

Two extensions have been added to format processing to make the writing of CSV (*comma-separated values*) files easier.

The first extension is the `g0` edit descriptor; 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 Fortran 2008 standard forbids the use of `g0.`*d* for anything other than floating-point data, even though *d* is ignored for `g`*w*`.`*d* for those data type, removing much of its convenience at a stroke.

Under the second extension, 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. It behaves similarly to *N*`(`*format-items*`)` for a very large integer *N*, for example

```
print '(4x,"List: ",*(g0,:,","))', 10, 20, 30
```

will print

```
    List: 10,20,30
```

However, due to a wording flaw in the published standard, the behaviour might differ if there is no colon between the last data edit descriptor and the closing right parenthesis of the unlimited format control. This flaw is expected to be corrected in due course, but we recommend that the situation should be avoided – that is, there should be a colon between the last data edit descriptor and the closing parenthesis.

20.8 Intrinsic procedures

There are a number of extensions to some existing intrinsic procedures, and a large number of new intrinsic procedures have been added. This highlights the need for programmers to use explicit interfaces (and where possible, module procedures) to avoid inadvertant changes in their programs' semantics following an upgrade to a new language level in their compiler.

In the new intrinsic procedures described in this chapter, all arguments are intent `in` unless otherwise specified. Any argument named `kind` must (if present) be a scalar integer initialization expression.

20.9 Mathemetical intrinsic functions

20.9.1 Changes to trigonometric functions

The intrinsic functions `acos`, `asin`, `atan`, `cosh`, `sinh`, `tan`, and `tanh` now accept arguments of type complex. In the case of `cosh`, `sinh`, `tan`, and `tanh`, these were previously available by using the simple identities

$$\begin{aligned}
\cosh x &= \cos ix \\
\sinh x &= -i \sin ix \\
\tan x &= (\sin x)/(\cos x) \\
\tanh x &= -i \tan ix
\end{aligned}$$

or, by using Fortran statement functions (see Appendix C.1.5),

```
complex   :: cosh, sinh, tan, tanh, x
intrinsic :: cos, sin
cosh(x) = cos((0,1)*x)
sinh(x) = (0,-1)*sin((0,1)*x)
tan(x)  = sin(x)/cos(x)
tanh(x) = (0,-1)*sin((0,1)*x)/cos((0,1)*x)
```

20.9.2 New hyperbolic trigonometic functions

Elemental intrinsic functions have been added for the inverse hyperbolic trigonometric functions:

`acosh (x)` returns the inverse hyperbolic cosine of `x`, that is, `y` such that `cosh(y)` would be approximately equal to `x`.

`asinh (x)` returns the inverse hyperbolic sine of `x`.

`atanh (x)` returns the inverse hyperbolic tangent of `x`.

In each case, `x` must be of type real or complex, and the result has the same type and kind.

Note that for complex numbers, these functions are related to the normal trigonometric functions by simple identities:

$$\begin{aligned}
\text{acosh}\, x &= -i \,\text{acos}\, x \\
\text{asinh}\, x &= -i \,\text{asin}\, ix \\
\text{atanh}\, x &= \text{atan}\, ix
\end{aligned}$$

20.9.3 New special mathematical functions

New elemental intrinsic functions for calculating Bessel functions have been added:

`bessel_j0 (x)` first kind and order zero;

`bessel_j1 (x)` first kind and order one;

`bessel_jn (n, x)` first kind and order `n`;

`bessel_y0 (x)` second kind and order zero;

`bessel_y1 (x)` second kind and order one;

`bessel_yn (n, x)` second kind and order `n`.

In each case, `x` must be of type real, and `n` must be of type integer with a non-negative value.
Two new transformational functions return vectors 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.
Three new elemental functions have been added for calculating the error function.

`erf (x)` returns the value of the error function of `x`, $\frac{2}{\sqrt{\pi}} \int_0^{\mathrm{x}} e^{-t^2}\, dt$.

`erfc (x)` returns the complement of the error function, 1−`erf(x)`. This has the mathematical form $\frac{2}{\sqrt{\pi}} \int_{\mathrm{x}}^{\infty} e^{-t^2}\, dt$.

`erfc_scaled (x)` returns the exponentially scaled error function, `exp(x**2)*erfc(x)`.

In each case, `x` must be of type real. Note that for small values of `x` (approximately 9.0 for IEEE single precision), `erf(x)` is equal to one and `erfc(x)` underflows to zero; when working outside this range, `erfc_scaled` is more useful.
Two new elemental functions have been added for the gamma function:

`gamma (x)` returns the value of the gamma function at `x`.

`log_gamma (x)` returns the natural logarithm of the absolute value of the gamma function, `log(abs(gamma(x)))`.

In each case, `x` must be of type real, with a value that is not zero or a negative whole number.

20.9.4 Euclidean norms

Two new functions have been added for calculating Euclidean norms (or distance). The first function is elemental:

`hypot (x, y)` returns the Euclidean distance, that is $\sqrt{x^2+y^2}$, 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. This addition to the list of intrinsic functions makes a total of 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))` and `norm2([x, y])` – the former being available for this purpose since Fortran 90.)

The second function is transformational:

`norm2 (x` *[*`, dim`*]*`)` returns the L_2 norm of a real array `x`; the result is a real scalar of the same kind as `x`.

The L_2 norm is the square root of the sum of the squares of the elements; for a real vector this is mathematically equal (but not necessarily computationally equal) to `sqrt(dot_product(x, x))`.

If the `dim` argument is present it must be a scalar integer satisfying $1 \leq$`dim`$\leq n$, where n is the rank of `x`; it is not permitted to be an optional dummy argument. In this case, instead of calculating the L_2 norm of the whole array, the array is reduced exactly as for `sum(x, dim)` except that instead of summation the values are calculated by applying `norm2` to the vectors being reduced.

For example,

```
norm2(reshape([1.0, 3.0, 2.0, 4.0], [2, 2]), dim=2)
```

is approximately equal to `[2.236, 5.0]`.

The standard recommends, but does not require, that `norm2` be calculated without undue overflow or underflow.

20.10 Bit manipulation

A wide range of new intrinsic functions provide additional bit manipulation functionality.

20.10.1 Bitwise (unsigned) comparison

Four new elemental functions have been 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 binary, octal, or hexadecimal ('boz') literal constants; if of type integer, they need not have the same kind type parameter. For example, on a two's-complement processor, `-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.

20.10.2 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.

The arguments `i` and `j` must be either 'boz' literal constants or of type integer; if they are of type integer they must have the same kind, if one is a 'boz' constant it will be converted to the type of the other as if by the `int` function. They cannot both be 'boz' constants. The `shift` argument must be an integer, but can be of any kind. The result type is integer with the same kind as either `i` or `j`. An example is `dshiftl(21_int8, 64_int8, 2)`, which 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.

20.10.3 Bitwise reductions

Three new transformational functions reduce an array (by one rank, or completely to a scalar, in exact analogy to `sum` and `product`), but using bitwise operations instead of addition or multiplication (see Section 8.11).

iall (array *[*, mask*]*) reduces `array` to a scalar value using the `iand` function.

iall (array, dim *[*, mask*]*) reduces dimension `dim` of `array` using the `iand` function.

iany (array *[*, mask *]*) reduces `array` to a scalar value using the `ior` function.

iany (array, dim *[*, mask *]*) reduces dimension `dim` of `array` using the `ior` function.

iparity (array *[*, mask *]*) reduces `array` to a scalar value using the `ieor` function.

iparity (array, dim *[*, mask *]*) reduces dimension `dim` of `array` using the `ieor` function.

The `array` argument must be an integer array. If present, the `mask` argument must be a logical array with the same shape as `array`, and only those elements for which `mask` is true contribute to the final value. The `dim` argument must be an integer scalar in the range 1 ≤ `dim` ≤ rank(`array`); it must not be an optional dummy argument. The result of each function is of type integer with the same kind as `array`, and is either scalar or, if `dim` is specified, has the shape of `array` with that dimension eliminated.

If no elements contribute to the result or to an element of the result, the value is zero for `iany` and `iparity`, and `not(0_k)` for `iall`, where `k` = `kind(array)`.

For example, the value of `iall([14, 13, 11])` is equal to `8`, and the value of `iall([14, 13, 11], mask=[.true., .false., .true])` is equal to `10`.

20.10.4 Counting bits

Several new 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 parity of the bit count of an integer, that is, `poppar(i)` is identical to `iand(popcnt(i), 1)`.

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, `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.

20.10.5 Producing bitmasks

New 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. This has the same value as `ishft(1, i) - 1` on two's complement machines (all modern computers).

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≤`i`≤*b*, where *b* is the bit size of the result.

For example, `maskl(3,int8` is equal to `int(b'11100000',int8)` and `maskr(3,int8)` is equal to `7_int8`.

20.10.6 Merging bits

A new 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)))`.

20.10.7 Additional shift operations

There are three new intrinsic functions for bit shifting:

`shifta (i, shift)` returns the bits of `i` shifted right by `shift` bits, but instead of shifting in zero bits from the left, the 'sign bit' is replicated. On a two's complement machine this makes it an arithmetic shift (thus the name `shifta`), that is division by a power of two; in the unlikely event of encountering a one's complement or sign-magnitude machine, the interpretation of the result value is somewhat different.

`shiftl (i, shift)` returns the bits of `i` shifted left, equivalent to `ishft(i, shift)`;

`shiftr (i, shift)` returns the bits of `i` shifted right, equivalent to `ishft(i, -shift)`.

In each case, the `i` and `shift` are both of type integer (of any kind), `shift` must be in the range 0≤`shift`≤`bit_size(i)`, and the result is of type integer with the same kind as `i`.

The only 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).

20.11 Miscellaneous intrinsic procedures

20.11.1 Procedures supporting coarrays

The intrinsic subroutines `atomic_define` and `atomic_ref`, and the intrinsic functions `image_index`, `lcobound`, `num_images`, `this_image`, and `ucobound` have been added to support programming with coarrays. These are described in Chapter 19.

20.11.2 Executing another program

The ability to execute another program from within a Fortran program is provided by the new 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 that, 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.

20.11.3 Character comparison

In the unlikely event of the compiler supporting an ASCII character kind but it not being the default character kind, the intrinsic functions for comparing characters using the ASCII collating sequence `lge`, `lgt`, `lle`, and `llt` will also accept arguments of ASCII kind. Both arguments must have the same kind.

This is unlikely to be useful since ordinary comparison using the relational operators $<$ etc. has exactly the same results with ASCII kind.

20.11.4 Array searching

The transformational intrinsic functions `maxloc` and `minloc`, which search an array for the maximum or minimum value, respectively, have had an optional `back` argument added to indicate whether the first or last occurrence is desired. The `back` argument is the final argument in the list, and 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`.

The new transformational intrinsic function `findloc` reduces an array in exactly the same way as `maxloc` or `minloc`, but returning the position of an element with the specified value instead of the maximum or minimum – or zero if no element was 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 with the same shape as `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 dimention `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.

20.11.5 Logical parity

The new transformational intrinsic function `parity` reduces the array `mask`, which must be of type logical, in the same way as `all` or `any`, but with the `.neqv.` operation instead of the `.and.` or `.or.` operation. It has the form

`parity (mask)` reduces `mask` to a scalar, and

parity (mask, dim) reduces dimension `dim` of `mask`.

Its value is identical to that of `iand(count(mask [, dim]),1)==1`, that is, it tests whether the number of true values is odd, but it is possibly more efficient as well as clearer.

Note that, as in `count`, the actual argument corresponding to `dim` must not itself be an optional dummy argument.

20.11.6 Decimal arithmetic support

In order to facilitate support of non-binary arithmetics, in particular the decimal arithmetic specified by the 2008 IEEE 754 floating-point standard, an optional `radix` argument has been added to the inquiry function `selected_real_kind` at the end of its argument list. It must be a scalar integer, and limits the values returned to floating-point types with that radix; for example,

```
selected_real_kind(p=6, radix=10)
```

will return the kind parameter of a decimal floating-point type with at least six digits of precision if one is available. A return value of -5 indicates that the processor has no real kind with that radix.

The `radix` argument has also been added to the `ieee_selected_real_kind` function from the `ieee_arithmetic` module, with similar semantics.

20.11.7 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 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.

20.12 Additions to the iso_fortran_env module

20.12.1 Compilation information

Two inquiry functions have been added to this module 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'`.

20.12.2 Names for common kinds

Named constants for some frequently desired kind values for integer and real types have been added, these are:

`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

```
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
```

the use of `int64` allows this 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 20.14.

Figure 20.14 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
```

20.12.3 Kind arrays

Named array constants containing all the kind type parameter values for intrinsic types that are supported by the processor have been added. The named constants `character_kinds`, `integer_kinds`, `logical_kinds`, and `real_kinds` contain the supports 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.

20.12.4 Coarray support facilities

The module also contains the derived type `lock_type`, and the named constants `atomic_int_kind`, `atomic_logical_kind`, `stat_locked_other_image`, `stat_stopped_image`, and `stat_unlocked`. These are all described in Chapter 19.

20.13 Changes to other standard intrinsic modules

20.13.1 The iso_c_binding module

The inquiry function `c_sizeof` has been added to the intrinsic module `iso_c_binding`. 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 Chapter 12), and is not permitted to be an assumed-size array (see Section B.3).

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.

20.13.2 The ieee_arithmetic module

For the inquiry function `ieee_selected_real_kind`, an optional argument `radix` has been added at the end of the argument list. It must be a scalar integer, and limits the values returned to floating-point types with that radix; for example,

```
ieee_selected_real_kind(p=6, radix=10)
```

will return the kind parameter of an IEEE decimal floating-point type with at least six digits of precision if one is available. A return value of -5 indicates that the processor has no IEEE real type with that radix.

The `radix` argument has also been added to the intrinsic `selected_real_kind` function, with similar semantics (see Section 20.11.6).

20.14 Programs and procedures

20.14.1 Saved module entities

Variables and procedure pointers declared in the specification part of a module now implicitly have the `save` attribute; this may be confirmed by explicit specification. That is, in

```
module saved2008
  real :: x
  real, save :: y
end module
```

the variables `x` and `y` both have the `save` attribute. This means that they will retain their values even when no procedure is referencing the module; in Fortran 2003, the variable `x` would have become undefined at such a time.

Furthermore, compilers were permitted to deallocate unsaved allocatable variables in a module when the module was not being referenced. As there were no Fortran compilers which actually took advantage of this licence to deallocate the memory of such variables (so that in practice all module variables were effectively saved anyway), allowing the user to rely on the variables being saved is a useful simplification of the language.

20.14.2 Automatic pointer targetting

An actual argument with the `target` attribute is now permitted to correspond to a dummy pointer with the intent `in` attribute. This is illustrated by Figure 20.15; in this case, the

Figure 20.15 Convenient automatic targetting.

```
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
```

automatic 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 20.4.3). 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 20.16, 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 20.16 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)
```

20.14.3 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 20.17 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 20.18.

Figure 20.17 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
```

Figure 20.18 Huge unreadable nested if.

```
    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.)
```

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` might have different mixtures of `allocatable` and `pointer`, or indeed wish to pass plain variables.

Exercises

1. 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.

2. 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

$$x_{new} = mod(16807x_{old}, 2^{31} - 1).$$

This page intentionally left blank

A. Intrinsic procedures

In this appendix, we list all the intrinsic procedures, giving the names of their arguments and a short description. Where a procedure or procedure argument was added in Fortran 2003, we place a superscript 3 after its name. Where a procedure or procedure argument was added in Fortran 2008, we place a superscript 8 after its name.

The names of all the intrinsic procedures are included in the Index, which can therefore be used to find the full descriptions.

Name	Description
`abs (a)`	Absolute value.
`achar (i [,kind`^{3}`])`	Character in position `i` of ASCII collating sequence.
`acos (x)`	Arc cosine (inverse cosine) function.
`acosh`^{8} `(x)`	Inverse hyperbolic cosine function.
`adjustl (string)`	Adjust left, removing leading blanks and inserting trailing blanks.
`adjustr (string)`	Adjust right, removing trailing blanks and inserting leading blanks.
`aimag (z)`	Imaginary part of complex number.
`aint (a [,kind])`	Truncate to a whole number.
`all (mask [,dim])`	True if all elements are true.
`allocated (array)` or	True if the array is allocated.
`allocated`^{3} `(scalar)`	True if the scalar is allocated.
`anint (a [,kind])`	Nearest whole number.
`any (mask [,dim])`	True if any element is true.
`asin (x)`	Arcsine (inverse sine) function.
`asinh`^{8} `(x)`	Inverse hyperbolic sine function.
`associated (pointer [,target])`	True if pointer is associated with target.
`atan (x)`	Arctangent (inverse tangent) function.
`atan`^{8} `(y, x)`	Argument of complex number `(x, y)`.
`atanh`^{8} `(x)`	Inverse hyperbolic tangent function.
`atan2 (y, x)`	Argument of complex number `(x, y)`.
`call atomic_define`^{8} `(atom, value)`	Define `atom` atomically with the value `value`.
`call atomic_ref`^{8} `(value, atom)`	Define `value` atomically with the value of `atom`.

bessel_j0[8] (x)	Bessel function of the first kind and order zero.
bessel_j1[8] (x)	Bessel function of the first kind and order one.
bessel_jn[8] (n, x)	Bessel function of the first kind and order n.
bessel_jn[8] (n1, n2, x)	Bessel functions of the first kind, orders n1 to n2.
bessel_y0[8] (x)	Bessel function of the second kind and order zero.
bessel_y1[8] (x)	Bessel function of the second kind and order one.
bessel_yn[8] (n, x)	Bessel function of the second kind and order n.
bessel_yn[8] (n1, n2, x)	Bessel functions of the second kind, orders n1 to n2.
bge[8] (i, j)	True if i is bitwise greater than or equal to j.
bgt[8] (i, j)	True if i is bitwise greater than j.
ble[8] (i, j)	True if i is bitwise less than or equal to j.
blt[8] (i, j)	True if i is bitwise less than j.
bit_size (i)	Maximum number of bits that may be held in an integer.
btest (i, pos)	True if bit pos of integer i has value 1.
ceiling (a *[, kind]*)	Least integer greater than or equal to its argument.
char (i *[,kind]*)	Character in position i of the processor collating sequence.
cmplx (x *[,y] [,kind]*)	Convert to complex type.
command_argument_count[3] ()	Number of command arguments.
conjg (z)	Conjugate of a complex number.
cos (x)	Cosine function.
cosh (x)	Hyperbolic cosine function.
count (mask *[,dim] [,kind[3]]*)	Number of true elements.
call cpu_time (time)	Processor time.
cshift (array, shift *[,dim]*)	Perform circular shift.
call date_and_time (*[date] [,time] [,zone] [,values]*)	Real-time clock reading date and time.
dble (a)	Convert to double precision real.
digits (x)	Number of significant digits in the model for x.
dim (x, y)	max(x-y, 0).
dot_product (vector_a, vector_b)	Dot product.
dprod (x, y)	Double precision real product of two default real scalars.
dshiftl[8] (i, j, shift)	Most significant half of a double-width left shift.
dshiftr[8] (i, j, shift)	Least significant half of a double-width right shift.
eoshift (array, shift *[,boundary] [,dim]*)	Perform end-off shift.
epsilon (x)	Number that is almost negligible compared with one in the model for numbers like x.
erf[8] (x)	Error function.
erfc[8] (x)	Complementary error function.
erfc_scaled[8] (x)	Scaled complementary error function.

call execute_command_line[8] (command [, wait] [,exitstat] [,cmdstat] [,cmdmsg])	Execute command line.
exp (x)	Exponential function.
exponent (x)	Exponent part of the model for x.
extends_type_of[3] (a, mold)	Type extension inquiry.
findloc[8] (array, value [,mask] [,kind] [,back]) or findloc[8] (array, value, dim [,mask] [,kind] [,back])	Find the location in array of an element with value value.
floor (a [,kind])	Greatest integer less than or equal to its argument.
fraction (x)	Fractional part of the model for x.
gamma[8] (x)	Gamma function.
call get_command[3] ([command] [,length] [,status])	Get command line.
call get_command_argument[3] (number [, value] [,length] [,status])	Get single command argument.
call get_environment_variable[3] (name [, value] [,length] [,status] [,trim_name])	Get environment variable.
huge (x)	Largest number in the model for numbers like x.
hypot[8] (x, y)	Euclidean distance function $\sqrt{x^2+y^2}$.
iachar (c [,kind[3]])	Position in ASCII collating sequence.
iall[8] (array, dim [,mask]) or iall[8] (array [,mask])	Perform bitwise and operations.
iand (i, j)	Logical and on the bits.
iany[8] (array, dim [,mask]) or iany[8] (array [,mask])	Perform bitwise or operations.
ibclr (i, pos)	Clear bit pos to zero.
ibits (i, pos, len)	Extract a sequence of bits.
ibset (i, pos)	Set bit pos to one.
ichar (c [,kind[3]])	Position in the processor collating sequence.
ieor (i, j)	Exclusive or on the bits.
image_index[8] (coarray, sub)	Index of the image given by the cosubscripts sub for coarray.
index (string, substring [,back] [,kind[3]])	Starting position of substring within string.
int (a [,kind])	Convert to integer type.
ior (i, j)	Inclusive or on the bits.
iparity[8] (array, dim [,mask]) or iparity[8] (array [,mask])	Perform bitwise exclusive or operations.
is_contiguous[8] (array)	True if array is contiguous.

`ishft (i, shift)`	Logical shift on the bits.
`ishftc (i, shift [,size])`	Logical circular shift on a set of bits on the right.
`is_isostat_end`[3] `(i)`	Test value for end-of-file condition.
`is_isostat_eor`[3] `(i)`	Test value for end-of-record condition.
`kind (x)`	Kind type parameter value.
`lbound (array [,dim] [,kind`[3]`])`	Array lower bounds.
`lcobound`[8] `(coarray [,dim] [,kind])`	Coarray lower cobounds.
`leadz`[8] `(i)`	Number of leading zero bits in `i`.
`len (string [,kind`[3]`])`	Character length.
`len_trim (string [,kind`[3]`])`	Length of string without trailing blanks.
`lge (string_a, string_b)`	ASCII greater than or equal.
`lgt (string_a, string_b)`	ASCII greater than.
`lle (string_a, string_b)`	ASCII less than or equal.
`llt (string_a, string_b)`	ASCII less than.
`log (x)`	Natural (base *e*) logarithm function.
`log_gamma`[8] `(x)`	Logarithm of absolute value of gamma function.
`log10 (x)`	Common (base 10) logarithm function.
`logical (l, [,kind])`	Convert between kinds of logicals.
`maskl`[8] `(i [,kind])`	Integer with leftmost `i` bits 1 and the rest 0.
`maskr`[8] `(i [,kind])`	Integer with rightmost `i` bits 1 and the rest 0.
`matmul (matrix_a, matrix_b)`	Matrix multiplication.
`max (a1, a2 [,a3,...])`	Maximum value.
`maxexponent (x)`	Maximum exponent in the model for reals like `x`.
`maxloc (array [,mask] [,kind`[3]`] [,back`[8]`])` or `maxloc (array, dim [,mask] [,kind`[3]`] [,back`[8]`])`	Location of maximum array element.
`maxval (array [,mask])` or `maxval (array, dim [,mask])`	Value of maximum array element.
`merge (tsource, fsource, mask)`	`tsource` when `mask` is true; `fsource` otherwise.
`merge_bits`[8] `(i, j, mask)`	Merge the bits of `i` and `j` under control of `mask`.
`min (a1, a2 [,a3,...])`	Minimum value.
`minexponent (x)`	Minimum exponent in the model for reals like `x`.
`minloc (array [,mask] [,kind`[3]`] [,back`[8]`])` or `minloc (array, dim [,mask] [,kind`[3]`] [,back`[8]`])`	Location of minimum array element.
`minval (array [,mask])` or `minval (array, dim [,mask])`	Value of minimum array element.
`mod (a, p)`	Remainder modulo `p`, that is `a-int(a/p)*p`.
`modulo (a, p)`	`a` modulo `p`.
`call move_alloc`[3] `(from, to)`	Move allocation.
`call mvbits (from, frompos, len, to, topos)`	Copy bits.

`nearest (x, s)`	Nearest different machine number in the direction given by the sign of `s`.
`new_line`[3] `(a)`	Newline character.
`nint (a [,kind])`	Nearest integer.
`norm2`[8] `(x [,dim])`	Euclidean vector norm.
`not (i)`	Logical complement of the bits.
`null([mold])`	Disassociated pointer.
`num_images`[8] `()`	Number of images.
`pack (array, mask [,vector])`	Pack elements corresponding to true elements of `mask` into rank-one result.
`parity`[8] `(mask [,dim])`	True if number of true values is odd.
`popcnt`[8] `(i)`	Number of one bits in `i`.
`poppar`[8] `(i)`	1 if `popcnt(i)` is odd or 0 otherwise.
`precision (x)`	Decimal precision in the model for `x`.
`present (a)`	True if optional argument is present.
`product (array [,mask])` or `product (array, dim [,mask])`	Product of array elements.
`radix (x)`	Base of the model for numbers like `x`.
`call random_number (harvest)`	Random numbers in range $0 \le x < 1$.
`call random_seed ([size] [put] [get])`	Initialize or restart random number generator.
`range (x)`	Decimal exponent range in the model for `x`.
`real (a [,kind])`	Convert to real type.
`repeat (string, ncopies)`	Concatenates `ncopies` of `string`.
`reshape (source, shape [,pad] [,order])`	Reshape `source` to shape `shape`.
`rrspacing (x)`	Reciprocal of the relative spacing of model numbers near `x`.
`same_type_as`[3] `(a, b)`	Compare dynamic types.
`scale (x, i)`	$x \times b^i$, where b=`radix(x)`.
`scan (string, set [,back] [,kind`[3]`])`	Index of leftmost (rightmost if `back` is true) character of `string` that is in `set`; zero if none.
`selected_char_kind`[3] `(name)`	Kind of character set called `name`.
`selected_int_kind (r)`	Kind of type parameter for specified exponent range.
`selected_real_kind ([p] [,r] [,radix`[8]`])`	Kind of type parameter for specified precision and exponent range.
`set_exponent (x, i)`	Model number whose sign and fractional part are those of `x` and whose exponent part is `i`.
`shape (source [,kind`[3]`])`	Array (or scalar) shape.
`shifta`[8] `(i, shift)`	Shift the bits of `i` right by `shift`, filling with the leftmost bit.
`shiftl`[8] `(i, shift)`	Shift the bits of `i` left by `shift`, filling with 0.
`shiftr`[8] `(i, shift)`	Shift the bits of `i` right by `shift`, filling with 0.

`sign (a, b)`	Absolute value of `a` times sign of `b`.
`sin (x)`	Sine function.
`sinh (x)`	Hyperbolic sine function.
`size (array [,dim] [,kind`[3]`])`	Array size.
`spacing (x)`	Absolute spacing of model numbers near `x`.
`spread (source, dim, ncopies)`	`ncopies` copies of `source` forming an array of rank one greater.
`sqrt (x)`	Square-root function.
`storage_size`[8] `(a, [kind])`	Storage size in bits.
`sum (array [,mask])` or `sum(array, dim [,mask])`	Sum of array elements.
`call system_clock ([count] [,count_rate] [,count_max])`	Integer data from real-time clock.
`tan (x)`	Tangent function.
`tanh (x)`	Hyperbolic tangent function.
`this_image`[8] `()` or	Index of the invoking image.
`this_image`[8] `(coarray [,dim])`	Cosubscripts of `coarray` that denote data on the invoking image.
`tiny (x)`	Smallest positive number in the model for numbers like `x`.
`trailz`[8] `(i)`	Number of trailing zero bits in `i`.
`transfer (source, mold [,size])`	Same physical representation as `source`, but type of `mold`.
`transpose (matrix)`	Matrix transpose.
`trim (string)`	Remove trailing blanks from a single string.
`ubound (array [,dim] [,kind`[3]`])`	Array upper bounds.
`ucobound`[8] `(coarray [,dim] [,kind])`	Coarray upper cobounds.
`unpack (vector, mask, field)`	Unpack elements of `vector` corresponding to true elements of `mask`.
`verify (string, set [,back] [,kind`[3]`])`	Zero if all characters of `string` belong to `set` or index of leftmost (rightmost if `back` true) that does not.

B. Deprecated features

B.1 Introduction

This appendix describes features that are redundant within Fortran 95 and whose use we deprecate. 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.

B.2 Storage association

B.2.1 Storage units

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 B.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 contains a `sequence` statement making it a *sequence type*:

```
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. In Fortran 2003, the type is allowed to have type parameters (Section 13.4) but no type-bound procedures (Section 14.6) are permitted.

A sequence type is a *numeric sequence type* if it has no type parameters, 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 it has no type parameters, 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 combination of type and type parameters.

A `private` statement may be added to a sequence type definition, making its components private. The `private` and `sequence` statements may be interchanged but must be the second and third statements of the type definition.

Two 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. However, 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.

B.2.2 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

[1] If one or both types have been accessed by use association and renamed, it is the original names that must agree.

```
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 B.2.1). 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.

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 the `equivalence` was put are replaced by automatic arrays, allocatable arrays, pointers (reuse of storage, Sections 6.4 and 6.5), pointers as aliases (storage mapping, Section 6.15), and the `transfer` function (mapping of one data type onto another, Section 8.9).

B.2.3 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 routines 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 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 a pointer array, 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 routines, 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 routine 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 routine. 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 is when the limit of 39 continuation lines is 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.2.4 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-stmt]...
`end` *[*`block data` *[block-data-name]]*

where each *specification-stmt* is an `implicit`, `use`, type declaration (including `double precision`), `intrinsic`, `pointer`, `target`, `common`, `dimension`, `data`, `equivalence`, `parameter`, or `save` statement or derived-type definition. A type declaration statement must not specify the `allocatable`, `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.2.5 Coarrays and storage association

Coarrays are not permitted in `common` and `equivalence` statements.

B.3 Shape and character length disagreement

In Fortran 77, it was often convenient, when passing an array, not to have to specify the size of the dummy array. For this case, the *assumed-size* array declaration is available, where the last *bounds* in the *bounds-list* is

[lower-bound:*]* `*`

and the other bounds (if any) must be declared explicitly. Such an array must not be a function result.

Since an assumed-size array has no bounds in its last dimension, it does not have a shape and, therefore, must not be used as a whole array in an executable statement, except as an argument to a procedure that does not require its shape. However, if an array section is formed with an explicit upper bound in the last dimension, this has a shape and may be used as a whole array.

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. This is because this would require an action for every element of an array of unknown shape.

An object of one size or rank may be passed to an explicit-shape or assumed-size dummy argument array that is of another size or rank. If an array element is passed to an array, the actual argument is regarded as an array with elements that are formed from the parent array from the given array element onwards, in array element order. Figure B.1 illustrates this.

Here, only the last 49 elements of `a` are available to `sub`, as the first array element of `a` which is passed to `sub` is `a(52)`. Within `sub`, this element is referenced as `b(1)`.

Figure B.1 Passing an array element to an array.

```
real a(100)
:
call sub (a(52), 49)
:
subroutine sub(b,n)
:
real b(n)
```

In the same example, it would also be perfectly legitimate for the declaration of `b` to be written as `real b(7, 7)` and for the last 49 elements of `a` to be addressed as though they were ordered as a 7×7 array. The converse is also true. An array dimensioned 10×10 in a calling subroutine may be dimensioned as a singly dimensioned array of size 100 in the called subroutine. Within `sub`, it is illegal to address `b(50)` in any way, as that would be beyond the declared length of `a` in the calling routine. In all cases, the association is by storage sequence, in array element order.

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 an array dummy argument, 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. For an assumed-size 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. To avoid copies of this kind, a scalar actual argument is permitted to be associated with an array only if the actual argument is an element of an array that is not polymorphic, an assumed-shaped array, an array pointer, or is a subobject of such an element.

In Fortran 2003, these rules on character length disagreement have been extended to include `character(kind=c_char)` (which will often be the same as default character) and to 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. Note also that only a scalar dummy argument may be associated with a scalar actual argument.

Assumed-shape arrays (Section 6.3) supplant this feature.

B.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.

B.5 Other forms of loop control

B.5.1 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.[2] 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
```

[2]The `continue` statement is not limited to being the last statement of a `do` construct; it may appear anywhere among the executable statements.

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. Additional, but also redundant, `do` syntax is described in Appendix C.1.8.

B.5.2 The do while

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:

`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 the *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 the `do while` entails are fully described in Chapter 10 of *Optimizing Supercompilers for Supercomputers*, M. Wolfe (Pitman, 1989).

B.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, it 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 8 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)`.

B.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 dummy integer
                                  ! 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 7.2 that we regard implicit typing as dangerous. Therefore, the use of the `dimension` statement is not recommended.

In Fortran 2008, there is the `codimension` statement for declaring coarrays (Chapter 19) 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, the 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.

B.8 Specific names of intrinsic procedures

While all of the intrinsic procedures are generic, some of the intrinsic functions also have *specific names* 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 subprogram, provided they are specified in an `intrinsic` statement (Section 8.1.3).

Table B.1. Specific intrinsic functions not available as actual arguments.

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
		`amax0`	Integer	Real
		`max1`	Real	Integer
min(`a1`,`a2`,...)	`min(a1,a2,...)`	`min0`	Integer	Integer
		`amin1`	Real	Real
		`dmin1`	Double	Double
		`amin0`	Integer	Real
		`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 available as actual arguments.

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
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

B.9 Non-default mapping for implicit typing

The default for implicit typing (Section 7.2) 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 type 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 B.2 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 7.13) 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.

B.10 Fortran 2008 deprecated features

B.10.1 The sync memory statement and atomic subroutines

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 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.

Figure B.2

```
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
```

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 *atomic subroutines*, a new class of intrinsic subroutine. An atomic subroutine 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. The effect of executing an atomic subroutine is as if the action on the argument `atom` occurs instantaneously, and thus does not overlap with other atomic actions that might occur asynchronously. To avoid performance loss, the ordering of interleaved actions on different atomic variables in different images is not defined by the Standard.

`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)`. If its kind is the same as that of `value` or its type is `logical`, it is given the value of `value`. Otherwise, it is given the value `int(value, atomic_int_kind)`.

`value` has intent `in` and is a scalar of the same type as `atom`.

`call atomic_ref (value, atom)` defines `value` atomically with the value of `atom`.

`value` has intent `out` and is a scalar of the same type as `atom`. If its kind is the same as that of `atom` or its type is `logical`, it is given the value of `atom`. Otherwise, it is given the value `int(atom, kind(value))`.

`atom` has intent `in` and is a scalar coarray or coindexed object. It has type `integer(atomic_int_kind)` or `logical(atomic_logical_kind)`.

For example, consider the code in Figure B.3, 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 19.13.1.

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.

All of the image control statements except `critical`, `end critical`, `lock`, and `unlock` include the effect of executing a `sync memory` statement.

Figure B.3 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
```

B.10.2 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.

B.10.3 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.

B.10.4 Redundant contains statement

The `contains` statement in a module, non-module program unit, or type definition is no longer required to be followed by a module procedure definition, internal procedure definition, or type-bound procedure declaration. For example,

```
module trivial
  logical :: ok = .true.
contains
  ! nothing
end module
```

is permitted. In each case, the appearance of a `contains` statement without any following definition or declaration has no effect.

The purported use of this feature is for automatic generation of Fortran program units where the automatic generator is not clever enough to omit the `contains` statement in the case where there is nothing to follow it with. We consider this feature to be confusing and do not recommend its use.

B.10.5 The end statement

The `function` and `subroutine` keywords are now optional on the `end` statement of a module or internal subprogram. For example,

```
module m25
contains
  subroutine orbital
    ...
  end
end module
```

This means that if you have an old Fortran 77 subprogram you can turn it into an internal or module subprogram by simple inclusion into a source file that has the same source form (that is, the obsolete fixed source form).

It also means that seeing a bare `end` no longer necessarily means you are seeing the end of a program unit (or interface body). It is our opinion that this feature adds nothing to the language, and that it is better for all `end` statements to specify what they are ending, for example `end module`.

B.10.6 Referencing `atan2` by the name `atan`

Breaking with 50 years of tradition and nearly every other programming language in existence, the two-argument form of arctangent can now be referenced by the name `atan` instead of `atan2`.

The name `atan2` will forever remain usable for this purpose (for backwards compatibility) so there is little to gain from this, other than user confusion ('why is `atan` being called with two arguments?') and compiler and language bloat.

C. Obsolescent features

C.1 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.

C.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 second fields of up to 20 consecutive lines. 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:

- confine statement labels to positions 1 to 5 and statements to positions 7 to 72;

[1] This limit is processor dependent if the line contains characters other than those of the default type.

- 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).

C.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 (sl1, sl2, sl3,...) [,] intexpr
```

where *sl1*, *sl2*, *sl3*, 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

```
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).

C.1.3 Character length specification 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
```

C.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.

C.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 subroutine, 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. 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 6.10) 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.

C.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 C.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 C.2, or such a declaration must be accessible by use or host association.

Figure C.1 A function whose result is of assumed character length.

```
function copy(word)
   character(len=*) copy, word
   copy = word
end function copy
```

Figure C.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 Fortran 90/95 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.

C.1.7 Arithmetic if statement

The arithmetic `if` provides 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 replaced by the `if` statement and construct (Section 4.2). Its general form is

`if (`*expr*`)` *sl1*, *sl2*, *sl3*

where *expr* is any scalar expression of type integer or real, and *sl1*, *sl2*, and *sl3* are the labels of statements in the same scoping unit. If the result obtained by evaluating *expr* is negative then the branch to *sl1* is taken, if the result is zero the branch to *sl2* is taken, and if the result is greater than zero the branch to *sl3* is taken. An example is

```
    if (p-q) 1,2,3
 1  p = 0.
    go to 4
 2  p = 1.
    q = 1.
    go to 4
 3  q = 0.
 4  ...
```

in which a branch to 1, 2, or 3 is taken depending on the value of `p-q`. The arithmetic `if` may be used as a two-way branch when two of the labels are identical:

```
    if (x-y) 1,2,1
```

C.1.8 Shared do-loop termination

A `do`-loop may be terminated on a labelled statement other than an `end do` or `continue`. Such a statement must be an executable statement other than `go to`, a `return` or an `end` statement of a subprogram, a `stop` or an `end` statement of a main program, `exit`, `cycle`, arithmetic `if`, or assigned `go to` statement. Nested `do`-loops may share the same labelled terminal statement, in which case all the usual rules for nested blocks hold, but a branch to the label must be from within the innermost loop. Thus, we may write a matrix multiplication as

```
    a(1:n, 1:n) = 0.
    do 1 i = 1, n
       do 1 j = 1, n
          do 1 l = 1, n
 1           a(i, j) = a(i, j) + b(i, l)*c(l, j)
```

Execution of a `cycle` statement restarts the loop without execution of the terminal statement.

This form of `do`-loop offers no additional functionality but considerable scope for unexpected mistakes.

C.1.9 Alternate return

When calling certain types of subroutines, it is possible that specific exceptional conditions will arise, which 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` *intexpr*

where *intexpr* 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 *intexpr* evaluates to 2, the second alternate `return` will be taken. If *intexpr* evaluates to a value which is less than 1, or greater than the number of alternate `returns` 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.

C.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 B.6), or default complex types, and they are treated as equivalenced. The `result` clause plays exactly the same rôle 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 C.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.

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.

Figure C.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
```

C.3 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 must be of default kind. It is not printed but interpreted as a *carriage control character*. If it is a blank, no action is taken, and it is good practice to insert a blank as the first character of each record, either explicitly as `' '` or using the `t2` edit descriptor (described in Section 9.12.4), in order to avoid inadvertent generation of spurious carriage control characters. This can happen when the first character in an output record is non-blank, and might occur, for instance, when printing integer values with the format `'(i5)'`. Here, all output values between −999 and 9999 will have a blank in the first position, but all others will generate a character there which may be used mistakenly for carriage control.

The carriage control characters defined by the standard are:

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

As a precaution, the first character of each record produced by list-directed and `namelist` output is a blank, unless it is the continuation of a delimited character constant.

In this context, we note that execution of a `print` statement does not imply that any printing will actually occur, and nor does execution of a `write` statement imply that printing will not occur.

C.4 Features deleted in Fortran 95

The features listed in this section were deleted from the Fortran 95 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 very long-term portability and to avoid unnecessary compiler warning messages. They are fully described in previous 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 is 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 9.4).

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.

This page intentionally left blank

D. Avoiding compilation cascades

When compiling a module, most compilers produce two files: an object file, and a separate file containing the information for `use` of the module (we will call the latter the `.mod` file, as that is the most common suffix in use for it).

A program unit that `uses` a module depends on the `.mod` file, not on the object file or source file, but a recompilation cascade arises from any change to the module because

i) the `.mod` file depends on the source file; and

ii) the compiler updates the `.mod` file when compiling the source file even if there are no changes to it.

To avoid this cascade one must both break the connection between the source file and the `.mod` file, and avoid updating the `.mod` file when there are no changes to it.

We will describe how to do this using the `make` tool. Firstly, the Makefile needs a description of how to generate the `.mod` file when it does not exist. This should be done by a recursive `make` invocation. Secondly, the compiler needs to be prevented from updating the `.mod` file when there are no changes. This can be done by using a shell script to wrap the compiler invocation. We will show how to do these with the NAGWare f95 compiler.

In the simplest case we have a program, `example.exe` with two source files, `one.f90` containing a module that is used by the other, `two.f90`. The Makefile is shown in Figure D.1, and the shell script is shown in Figure D.2 (`f95` is the name by which the compiler is invoked).

The shell script uses the `-M` and `-nomod` options which cause the compiler only to produce the `.mod` files or not to produce any `.mod` files, respectively.

In the slightly more complicated case that `one` uses another module `zero` and does not make everything from `zero` private, then `zero.mod` must appear as a dependency of `one.mod` (as well as of `one.o`), that is

```
one.mod: zero.mod
```

This technique can be used for most compilers that store the module information in a separate file; for some of them, the time of compilation is written into the `.mod` file, so a more intelligent tool must be used to compare the new and old `.mod` files – this information can usually be obtained from the compiler vendor or from the World-Wide Web.

Figure D.1

```
# Makefile for example.exe

# Compilation options:
# (the -nocheck_modtime option suppresses the compiler's check for the
# .mod file being outofdate - since we are using make, we don't need
# this check.)
F90FLAGS = -O -nocheck_modtime

# Linking options:
F90LINKFLAGS =

# The executable depends on all the object files:
example.exe: one.o two.o
        f95 $F90LINKFLAGS -o example.exe one.o two.o

# Use mf95 to avoid .mod file updates
one.o: one.f90
        mf95 one one.f90

# No module(s) in two.f90, so can just use f95 directly
two.o: two.f90 one.mod
        f95 two.f90

# If there is no .mod file, 'make one.f90' will make sure we
# have an uptodate source file, then compile it asking for the
# .mod file only to be produced (this is very quick).
one.mod:
        make one.f90
        f95 -M one.f90
```

Figure D.2

```
#!/bin/sh
if [ -f $1.mod ]; then
#   Just produce the .mod file...
    mkdir $1.tmp
    f95 -M $F90FLAGS -mdir $1.tmp $2
    cmp -s $1.tmp/$1.mod $1.mod
    if [ $? != 0 ]; then
#       Different .mod file contents, so update it
        mv $1.tmp/$1.mod $1.mod
    fi
    rm -r $1.tmp
#   Now produce the object file and don't produce a .mod file
    f95 $F90FLAGS -nomod $2
else
# .mod file does not exist, just compile
    f95 $F90FLAGS $2
fi
```

This page intentionally left blank

E. 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 2.13, 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 heterogenous 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, 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
    procedure, non_overridable :: delete
    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)
  class(anyitem), target  :: item
  class(anyitem), pointer :: temp
  temp => item
  call remove(item)
  deallocate (temp)
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) :: 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, ') = (', es22.16, ', ', es22.16, ')')
    type is (real(kind(0d0)))
      print 3, kind(v), v
3     format(1x, 'real(kind=', i0, ') = ', es22.16)
    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)
        item%prevptr => item
      else if (associated(list%firstptr, item)) then
        ! First item in list.
        list%firstptr => item%nextptr         ! first = next.
```

```
        item%prevptr%prevptr => item%nextptr  ! last%prev = item%next.
        item%nextptr%prevptr => item%prevptr  ! next%prev = last.
      end if
    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) :: 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 p%delete
  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
end program
```

F. Fortran terms

The following is a list of the principal technical terms used in this book, and their definitions. To facilitate reference to the standard, we have kept closely to the meanings used there. We make no reference to deprecated, obsolescent, or deleted features (Appendices B and C) in this appendix.

abstract interface set of procedure characteristics with dummy argument names

actual argument entity that appears in a procedure reference

allocatable having the `allocatable` attribute

array set of scalar data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern

- **array element** scalar individual element of an array
- **array pointer** array with the `pointer` attribute
- **array section** array subobject that is itself an array
- **assumed-shape array** non-allocatable non-pointer dummy argument array that takes its shape from its effective argument
- **assumed-size array** dummy argument array whose size is assumed from that of its effective argument
- **deferred-shape array** allocatable array or array pointer
- **explicit-shape array** array declared with explicit values for the bounds in each dimension of the array

ASCII character character whose representation method corresponds to ISO/IEC 646:1991

associate name name of construct entity associated with a selector of an `associate` or `select type` construct

associating entity (in a dynamically established association) the entity that did not exist prior to the establishment of the association

association inheritance association, name association, pointer association, or storage association.

argument association association between an effective argument and a dummy argument

construct association association between a selector and an associate name in an `associate` or `select type` construct

host association name association, other than argument association, between entities in a submodule or contained scoping unit and entities in its host

inheritance association association between the inherited components of an extended type and the components of its parent component

linkage association association between a variable with the `bind` attribute and a C global variable

name association argument association, construct association, host association, linkage association, or use association

pointer association association between a pointer and an entity with the `target` attribute

use association association between entities in a module and entities in a scoping unit or construct that references that module, as specified by a `use` statement

attribute property of an entity that determines its uses

automatic data object (also **automatic object**) non-dummy data object with a type parameter or array bound that depends on the value of a specification expression that is not a constant expression

base object (of a subobject) object designated by the leftmost *part-name*

binding type-bound procedure or final subroutine

binding name name given to a specific or generic type-bound procedure in the type definition

binding label default character value specifying the name by which a global entity with the `BIND` attribute is known to the companion processor

block sequence of executable constructs within an executable construct that is bounded by statements of the executable construct

bound (also **array bound**) limit of a dimension of an array

branch target statement statement whose statement label appears as a label in a `go to` statement, computed `go to` statement, `end=` specifier, `eor=` specifier, or `err=` specifier

C address value identifying the location of a data object or procedure either defined by the companion processor or which might be accessible to the companion processor (this is the concept that the C standard calls the address)

character context within a character literal constant or within a character string edit descriptor

characteristics (of a dummy argument) being a dummy data object or dummy procedure

coarray data entity that has nonzero corank

cobound bound (limit) of a codimension

codimension dimension of the pattern formed by a set of corresponding coarrays

coindexed object data object whose designator includes an image selector

collating sequence one-to-one mapping from a character set into the non-negative integers

component part of a derived type, or of an object of derived type, defined in a type declaration statement

> **direct component** one of the components, or one of the direct components of a non-pointer non-allocatable component
>
> **parent component** component of an extended type whose type is that of the parent type and whose components are inheritance associated with the inherited components of the parent type
>
> **subcomponent** (of a structure) direct component that is a subobject of the structure
>
> **ultimate component** a component that is of intrinsic type, a pointer, or allocatable; or an ultimate component of a non-pointer non-allocatable component of derived type

component order ordering of the non-parent components of a derived type that is used for intrinsic formatted input/output and structure constructors (where component keywords are not used)

conformable (of two data entities) having the same shape, or one being an array and the other being scalar

connected relationship between a unit and a file: each is connected if and only if the unit refers to the file

constant data object that has a value and which cannot be defined, redefined, or become undefined during execution of a program

> **literal constant** constant that does not have a name
>
> **named constant** named data object with the `parameter` attribute

construct entity entity whose identifier has the scope of a construct

constant expression expression satisfying the requirements specified in Section 7.4, thus ensuring that its value is constant

contiguous (array) having array elements in order that are not separated by other data objects, as specified in Section 20.4.2

contiguous (multi-part data object) that the parts in order are not separated by other data objects

corank number of codimensions of a coarray (zero for objects that are not coarrays)

cosubscript scalar integer expression in an image selector

data entity data object, result of the evaluation of an expression, or the result of the execution of a function reference

data object (also **object**) constant, variable, or subobject of a constant

decimal symbol character that separates the whole and fractional parts in the decimal representation of a real number in a file

declaration specification of attributes for various program entities (often this involves specifying the type of a named data object or specifying the shape of a named array object)

default initialization mechanism for automatically initializing pointer components to have a defined pointer association status, and non-pointer components to have a particular value

default-initialized (subcomponent) subject to a default initialization specified in the type definition for that component

definable capable of definition and permitted to become defined

defined (data object) has a valid value

defined (pointer) has a pointer association status of associated or disassociated

defined assignment assignment defined by a procedure

defined input/output input/output defined by a procedure and accessed via a `dt` edit descriptor

defined operation operation defined by a procedure

definition (of a data object) process by which the data object becomes defined

definition (of a derived type, enumeration, or procedure) specification of the type, enumeration, or procedure

descendant (module or submodule) submodule that extends that module or submodule or that extends another descendant thereof

designator name followed by zero or more component selectors, complex part selectors, array section selectors, array element selectors, image selectors, and substring selectors

complex part designator designator that designates the real or imaginary part of a complex data object, independently of the other part

object designator (also **data object designator**) designator for a data object (an object name is a special case of an object designator)

procedure designator designator for a procedure

disassociated (pointer association) pointer association status of not being associated with any target and not being undefined

disassociated (pointer) has a pointer association status of disassociated

dummy argument entity whose identifier appears in a dummy argument list in a function or subroutine, or whose name can be used as an argument keyword in a reference to an intrinsic procedure or a procedure in an intrinsic module

dummy data object dummy argument that is a data object

dummy function dummy procedure that is a function

effective argument entity that is argument associated with a dummy argument

effective item scalar object that is associated with an edit descriptor as a result of the rules for an input/output list

elemental independent scalar application of an action or operation to elements of an array or corresponding elements of a set of conformable arrays and scalars, or possessing the capability of elemental operation (combination of scalar and array operands or arguments combine the scalar operand(s) with each element of the array operand(s))

elemental assignment assignment that operates elementally

elemental operation operation that operates elementally

elemental operator operator in an elemental operation

elemental procedure elemental intrinsic procedure or procedure defined by an elemental subprogram

elemental reference reference to an elemental procedure with at least one array actual argument

elemental subprogram subprogram with the `elemental` prefix

explicit initialization initialization of a data object by a specification statement

explicit interface interface of a procedure that includes all the characteristics of the procedure and names for its dummy arguments

extent number of elements in a single dimension of an array

external file file that exists in a medium external to the program

external unit (also **external input/output unit**) entity that can be connected to an external file

file storage unit unit of storage in a stream file or an unformatted record file

final subroutine subroutine whose name appears in a `final` statement in a type definition, and which can be automatically invoked by the processor when an object of that type is finalized

finalizable (type) has a final subroutine or a non-pointer non-allocatable component of finalizable type

finalizable (non-pointer data entity) of finalizable type

finalization the process of calling final subroutines when variables are deallocated or otherwise cease to exist

function procedure that is invoked by an expression

generic identifier lexical token that identifies a generic set of procedures, intrinsic operations, and/or intrinsic assignments

host instance (of an internal procedure, or dummy procedure or procedure pointer associated with an internal procedure) instance of the host procedure that supplies the host environment of the internal procedure

host scoping unit (also **host**) the scoping unit immediately surrounding another scoping unit, or the scoping unit extended by a submodule

IEEE infinity IEEE standard conformant infinite floating-point value

IEEE NaN IEEE standard conformant floating-point datum that does not represent a number

image instance of a Fortran program

image index integer value identifying an image

image control statement statement that affects the execution ordering between images

implicit interface interface of a procedure that includes only whether it is a function and if so, the type and type parameters of its result

inclusive scope non-block scoping unit plus every block scoping unit whose host is that scoping unit or that is nested within such a block scoping unit (that is, inclusive scope is the scope as if `block` constructs were not scoping units)

inherit (for extended type) acquire entities (components, type-bounds procedures, and type parameters) through type extension from the parent type

inquiry function intrinsic function, or function in an intrinsic module, whose result depends on the properties of one or more of its arguments instead of their values

interface (of a procedure) name, procedure characteristics, dummy argument names, binding label, and generic identifiers

generic interface set of procedure interfaces identified by a generic identifier

specific interface interface identified by a non-generic name

interface block abstract interface block, generic interface block, or specific interface block

abstract interface block interface block with the `abstract` keyword; collection of interface bodies that specify named abstract interfaces

generic interface block interface block with a generic-spec; collection of interface bodies and procedure statements that are to be given that generic identifier

specific interface block interface block with no generic-spec or `abstract` keyword; collection of interface bodies that specify the interfaces of procedures

interoperable (Fortran entity) equivalent to an entity defined by or definable by the companion processor

intrinsic type, procedure, module, assignment, operator, or input/output operation defined in the Standard and accessible without further definition or specification, or a procedure or module provided by a processor but not defined in the Standard

standard intrinsic (procedure or module) defined in the Standard

nonstandard intrinsic (procedure or module) provided by a processor but not defined in the Standard

internal file character variable that is connected to an internal unit

internal unit input/output unit that is connected to an internal file

ISO 10646 character character whose representation method corresponds to UCS-4 in ISO/IEC 10646

keyword statement keyword, argument keyword, type parameter keyword, or component keyword

argument keyword word that identifies the corresponding dummy argument in an actual argument list

component keyword word that identifies a component in a structure constructor

statement keyword word that is part of the syntax of a statement

type parameter keyword word that identifies a type parameter in a type parameter list

lexical token keyword, name, literal constant other than a complex literal constant, operator, label, delimiter, comma, =, =>, `:`, `::`, `;`, or `%`

line sequence of zero or more characters

main program program unit that is not a subprogram, module, submodule, or block data program unit

masked array assignment assignment statement in a `where` statement or `where` construct

module program unit containing (or accessing from other modules) definitions that are to be made accessible to other program units

name identifier of a program consituent, formed according to the rules given in Section 2.7

operand data value that is the subject of an operator

operator intrinsic operator, defined unary operator, or defined binary operator

passed-object dummy argument dummy argument of a type-bound procedure or procedure pointer component that becomes associated with the object through which the procedure is invoked

pointer data pointer or procedure pointer

data pointer data entity with the `pointer` attribute

procedure pointer procedure with the `external` and `pointer` attributes

pointer assignment association of a pointer with a target, by execution of a pointer assignment or an intrinsic assignment statement for a derived-type object that has the pointer as a subobject

polymorphic (data entity) able to be of differing dynamics types during program execution

preconnected (file or unit) connected at the beginning of execution of the program

procedure entity encapsulating an arbitrary sequence of actions that can be invoked directly during program execution

dummy procedure procedure that is a dummy argument

external procedure procedure defined by an external subprogram or by means other than Fortran

internal procedure procedure defined by an internal subprogram

module procedure procedure that is defined by a module subprogram

pure procedure procedure declared or defined to be pure according to the rules in Section 6.10

type-bound procedure procedure that is bound to a derived type and referenced via an object of that type

processor combination of a computing system and mechanism by which programs are transformed for use on that computing system

processor dependent not completely specified in the Standard, having methods and semantics determined by the processor

program set of Fortran program units, and perhaps entities defined by means other than Fortran, that includes exactly one main program

program unit main program, external subprogram, module, submodule, or block data program unit

rank number of array dimensions of a data entity (zero for a scalar entity)

record sequence of values or characters in a file

record file file composed of a sequence of records

reference data object reference, procedure reference, or module reference

> **data object reference** appearance of a data object designator in a context requiring its value at that point during execution
>
> **function reference** appearance of the procedure designator for a function, or operator symbol in a context requiring execution of the function during expression evaluation
>
> **module reference** appearance of a module name in a `use` statement
>
> **procedure reference** appearance of a procedure designator, operator symbol, or assignment symbol in a context requiring execution of the procedure at that point during execution; or occurrence of defined input/output or derived-type finalization

result variable variable that returns the value of a function

saved having the `save` attribute

scalar data entity that can be represented by a single value of the type and that is not an array

scoping unit `block` construct, derived-type definition, interface body, program unit, or subprogram, excluding all nested scoping units in it

> **block scoping unit** scoping unit of a `block` construct

sequence set of elements ordered by a one-to-one correspondence with the numbers $1, 2, \ldots n$

shape array dimensionality of a data entity, represented as a rank-one array whose size is the rank of the data entity and whose elements are the extents of the data entity (thus the shape of a scalar data entity is an array with rank one and size zero)

simply contiguous (array designator or variable) satisfying the conditions specified in Section 20.4.3 (these conditions are simple ones which make it clear that the designator or variable designates a contiguous array)

size (array) total number of elements in the array

specification expression expression satisfying the requirements specified in Section 7.14, thus being suitable for use in specifications

specific name name that is not a generic name

standard-conforming program program that uses only those forms and relationships described in, and has an interpretation according to, the Standard

statement sequence of one or more complete or partial lines satisfying the rules of Section 2.4

executable statement statement that performs or controls one or more actions, excluding those in the specification-part of a `block` construct

nonexecutable statement statement that is not an executable statement

statement entity entity whose identifier has the scope of a statement or part of a statement

statement label (also **label**) unsigned positive number of up to five digits that refers to an individual statement

stream file file composed of a sequence of file storage units

structure scalar data object of derived type

structure component component of a structure

structure constructor syntax that specifies a structure value or creates such a value

submodule program unit that extends a module or another submodule

subobject portion of data object that can be referenced, and if it is a variable defined, independently of any other portion

subprogram function subprogram or subroutine subprogram

external subprogram subprogram that is not contained in a main program, module, submodule, or another subprogram

internal subprogram subprogram that is contained in a main program or another subprogram

module subprogram subprogram that is contained in a module or submodule but is not an internal subprogram

subroutine procedure invoked by a `call` statement, by defined assignment, or by some operations on derived-type entities

atomic subroutine intrinsic subroutine that performs an action on its `atom` argument atomically

target entity that is pointer associated with a pointer, entity on the right-hand side of a pointer assignment statement, or entity with the `target` attribute

transformational function intrinsic function, or function in an intrinsic module, that is neither elemental nor an inquiry function

type (also **data type**) named category of data characterized by a set of values, a syntax for denoting these values, and a set of operations that interpret and manipulate the values

abstract type type with the `abstract` attribute

declared type type that a data entity is declared to have, either explicitly or implicitly

derived type type defined by a type definition or by an intrinsic module

dynamic type type of a data entity at a particular point during execution of a program

extended type type with the `extends` attribute

extensible type type that does not have the `bind` attribute and which therefore may be extended using the `extends` clause

extension type (of one type with respect to another) is the same type or is an extended type whose parent type is an extension type of the other type

intrinsic type type defined by the Standard that is always accessible

numeric type one of the types integer, real, and complex

parent type (of an extended type) type named in the `extends` clause

type compatible compatibility of the type of one entity with respect to another for purposes such as argument association, pointer association, and allocation

type parameter value used to parameterize a type

assumed type parameter length type parameter that assumes the type parameter value from another entity (the other entity is

- the selector for an associate name;
- the *constant-expr* for a named constant of type character; or
- the effective argument for a dummy argument).

deferred type parameter length type parameter whose value can change during execution of a program and whose type parameter value is a colon

kind type parameter type parameter whose value is required to be defaulted or given by a constant expression

length type parameter type parameter whose value is permitted to be assumed, deferred, or given by a specification expression

type parameter inquiry syntax that is used to inquire the value of a type parameter of a data object

type parameter order ordering of the type parameters of a type used for derived-type specifiers

ultimate argument non-dummy entity with which a dummy argument is associated via a chain of argument associations

undefined (data object) does not have a valid value

undefined (pointer) does not have a pointer association status of associated or disassociated

unit (also **input/output unit**) means for referring to a file

unlimited polymorphic able to have any dynamic type during program execution

unsaved not having the `save` attribute

variable data entity that can be defined and redefined during execution of a program

> **local variable** variable in a scoping unit that is not a dummy argument or part thereof, is not a global entity or part thereof, and is not accessible outside that scoping unit
>
> **lock variable** scalar variable of the type `lock_type` that is defined in the intrinsic module `iso_fortran_env`

vector subscript section subscript that is an array

whole array array component or array name without further qualification

G. 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	
ABCDEFG*u*	*u* = unchanged
ABCD*bbuu*	*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 6.13.

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 8.

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 first described in Section 8.6.

Chapter 6

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
```

10.

```
type(stack) a
allocate (a%content(4))
a%index = 1
a%content = (/ 1, 2, 3, 4 /)
a = stack(2, (/a%content, 5, 6 /))
```

11.

```
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
```

12.

```
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.,0.,i=7,9) /), &
                     (/ 6,9/) ))
```

Chapter 7

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 8.13.3.

4.

Letter	Scoping unit: mod	outer	inner	fun
a,b	character(10,2)	—	—	—
c-e	real	—	—	—
f	real	—	—	real
g,h	real	—	—	—
i-n	integer	—	—	—
o-w	real	—	—	—
x	real	—	—	real
y	real	—	—	—
z	real	—	complex	—

5.

i) `type(person) boss = person('Smith', 48.7, 22)`

ii) a) This is impossible because a pointer component cannot be a constant.

b)
```
type(entry) current
data current%value, current%index /1.0, 1/
```

6.

All are constant expressions except for:

iv) because of the real exponent; and

viii) because of the pointer component.

Chapter 8

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 & 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.

Chapter 9

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.

```
character, dimension(3,3) :: tic_tac_toe
integer                   :: unit
:
write (unit, '(t1, 3a2)') tic_tac_toe
```

4.

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)`

5.

```
   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
```

Chapter 12

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
```

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 15

1.

```
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 16

1.

```
!
! 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
```

2.

```
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
```

Chapter 17

1.

```
   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
```

2.

```
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 19

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.

<table>
<tr><th>Code with bottlenecks</th><th>Code without bottlenecks</th></tr>
<tr><td><pre>k = this_image()
if (k<=nz) then
   do i = 1, nx
      a(i, 1:ny) = b(1:ny, k)[i]
   end do
end if</pre></td><td><pre>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</pre></td></tr>
</table>

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 20

1.

```
! 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
    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)
      type(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)
      type(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)
      type(realvec_t), intent(inout) :: vec
      real, pointer                  :: vector(:)
      vec%vcount = vec%vcount + 1
      vector => vec%value
    end function
    subroutine get_usage(this, ecount, vcount)
      type(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)  :: i, nerefs
  :
  call a%new(n)
  do i=1, n
    a%element(i) = ...
  end do
  print *,a%vector()
  :
  call a%get_usage(ecount=nerefs)
```

2.

```
! A pseudo-random 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) stop '?RANDOM_NUMBER: PUT is too small'
      seed = sum(put)
      if (seed<=0 .or. seed>m) call init_random
    else if (present(get)) then
      if (ubound(get)<1) stop '?RANDOM_NUMBER: GET is too small'
      get(1) = seed
      get(2:) = 0
    else
      call init_random
    end if
  end subroutine
end module
```

This page intentionally left blank

Index

a edit descriptor, 203, 204
abs, 163
abstract
  interface, 259
    block, 259
  type, 280
abstract keyword, 280
access= specifier, 216, 220, 326
achar, 166, 309
acos, 165, 376
acosh, 376
action= specifier, 217, 221
actual
  argument, **74–92**, 100, 405, 406, 421, 424, 425
  procedure, 82
adjustl, 167
adjustr, 167
advance= specifier, **199**, 209, 212
aimag, 163
aint, 163
alias, 124
all, 175
allocatable
  coarray, **340**
  component, 106, 354, 356
    of coarray, 341–342
  scalar, 294
allocatable attribute, **102**, 123, 144, 151
allocatable statement, 144
allocate statement, 28, **102–109**, 270, 291, 372
allocated, **176**, 293
allocation (sourced), 292
allocation status, **102**, 176
alphanumeric characters, 9
alternate return, 423
ampersand, 12, 18
anint, 163
ANSI, 2, 3
any, 175
argument, 67, **73**, 78–83, 388, 389, 424
  intent, 162
  list, **83**, 92, 402, 424
arithmetic if statement, 422
array, 23–29, **99–127**, 138, 409
  allocation, 102
  argument, 73, 100
  assignment, **48**, 111
  bounds, **23**, 28, 100
  constant, 136
  constructor, 25, 88, **125–126**, 135, 137, 297–298
  element, 119
    order, 24
  expression, 46–48
  function, 110
  section, 25, 27, **120**
  subobject, 120–123
  subscript, 26, 247
array-valued
  function, 105, **110**
  object, 25
ASCII standard, 19, 166, 309
asin, 165, 376
asinh, 376
assembler, 2
assembly language, 81
assigned go to, 427
assignment, 46

assignment statement, 33, **37–42**, 121, 135
assignment to allocatable array, 294
`assignment(=)`, 91, 92
`associate` construct, 271
associate name, 271
`associated`, 37, **163**
association (of pointer), **37**, 77, 79, 103, 104, 145, 358
assumed character length, **93**, 422
assumed derived type parameter, 258
assumed-shape array, 100
assumed-size array, 405, 406
`asynchronous` attribute, 322
  coarray, 343
`asynchronous` statement, 322
asynchronous input/output, 320–322
`asynchronous=` specifier, 320, 326
`atan`, 165, 376
`atan2`, 165, 314
`atanh`, 376
atomic subroutines, 347, 415
`atomic_define`, 415
`atomic_ref`, 415
attributes, 145, 219
automatic
  array, 100
  data object, 100
  targetting, 388

`b` edit descriptor, **202**
`backspace` statement, 214
batch mode, 191
`bessel_j0`, 377
`bessel_j1`, 377
`bessel_jn`, 377
`bessel_y0`, 377
`bessel_y1`, 377
`bessel_yn`, 377
`bge`, 378
`bgt`, 378
binary
  constant, 15, 312
  operation, 46
  operator, **33**, 39, 43
binding, 279
  label, 249, 252
`bind` attribute, 246, 249, 250, 252
`bind` statement, 250
bit, 126, 171
  manipulation procedure, 171
`bit_size`, 172
blank
  character, 10, 11, 18, 190, 199
  common, 402, 403
  field, 217
`blank=` specifier, 205, 217, 221, 326
`ble`, 378
block, 55
`block` data, 404, 405
`block` construct, **366**
`blt`, 378
`bn` edit descriptor, **205**, 217
bound, 23, 28, 103, 122, 136, 176
branch, 63
  target statement, 63
`btest`, 172
byte, 15–17, 19, 126
`bz` edit descriptor, **205**, 217

C array, 247
C function, 249, 251
C pointer, 245
C programming language, 243
C prototype, 251
C struct type, 246
C type, 244, 245
`c_alert`, 244
`c_associated` , 245
`c_backspace`, 244
`c_bool`, 244
`c_carriage_return`, 244
`c_char`, 244
`c_double`, 244
`c_double_complex`, 244
`c_f_pointer`, 245, 253
`c_f_procpointer`, 246
`c_float`, 244
`c_float_complex`, 244
`c_form_feed`, 244

c_funloc, 245
c_funptr, 245
c_horizontal_tab, 244
c_loc, 245, 253
c_long_double, 244
c_long_double_complex, 244
c_new_line, 244
c_null_char, 244
c_ptr, 245
c_sizeof, 387
c_vertical_tab, 244
call statement, 70
carriage control, 426
case construct, **57–58**
case (of characters), 9
case default statement, 58
ceiling, 164
char, 166
character, **9**, 18, 19, 26, 419
 assignment, **41**, 196
 constant, 195, 196
 context, 12
 expression, **41**, 186, 189
 function, 166
 literal constant, 17, 18
 set, **9**, 29, 33, 309
 storage
  association, 400
  unit, 399
 string, 205
 substring, 26
 variable, **26**, 186, 194, 204
 varying length, 5, 101
character statement, 21, **153**, 420
character_kinds, 386
child data transfer statement, 318
class attribute, 268
class default guard, 273
class default statement, 272
class is guard, 273
class is statement, 272
class keyword, 267
clone, 293
close statement, 215, 218, 219
cmplx, 164
coarray, **333–352**
 allocatable, **340**
 allocatable component, 342
 component, 342
 dummy argument, 338
 input/output, 349
 pointer component, 342
 procedure component, 342
 with allocatable component, 341
 with pointer component, 341
cobound, **335**
codimension, **335**
coercion, 35
coextent, **335**
coindexed object, **336**
collating sequence, 18, 19
colon editing, 208
command
 argument, 308
 line, 308
command_argument_count, 308
comment line, 12
commentary, 12, 419
common block, 251, 284, 402–405, 407
common statement, 402
compilation cascade, 332, **429**
compiler, 2, 50, 133
compiler_options, 385
compiler_version, 385
complex
 constant, 195, 196, 314
 exponentiation, 37
 literal constant, 17
 operand, 36
 parts, 365
 values, 203
 variable, 20
complex statement, 21, **152**
component selector, 22
computed go to, 420
concatenation, 41, 168
conditional compilation, 6
conformance
 (of arrays), 46
 (to Fortran standard), 3, 7

conjg, 164
connection (of file), 213
constant, 13
  expression, 134, **135**, 137, 163, 166, 168, 298
  subobject of a, 137
constraint, 8
construct name, 61
contains statement, **69–73**, 190
contiguity, 76
contiguous attribute, **361**
continuation
  line, **12**, 312, 403, 419
  mark, **12**, 18
continue statement, 407
copy-in copy-out, 77
corank, **335**
cos, 165, 376
cosh, 165, 376
count, 175, 313
CPU, 2, 181, 213
cpu_time, 181
critical section, **347**
cshift, 178
current record, 199
cycle statement, 60, 423

d edit descriptor, 408
data
  abstraction, 13
  base, 209
  structure, 21
  transfer, 211, 318
  type, 13, 14, 29
data statement, 15, **138**, 312, 404, 420
data statement, 353
date, 180
date_and_time, 180
dble, 408
dc edit descriptor, 311
dead code, 63
deallocate statement, 102, **104**
decimal
  comma, 311
  edit mode, 311
  point, 311
decimal= specifier, 311, 326
declared type, 268
deep copying, **108**, 358, 373
default
  character constant, **19**, 205
  initialization, 141
  real constant, 16
  real variable, 20
  type parameter value, 258
deferred
  binding, 280
  type parameter, 255, 258, 291
  type-bound procedure, 281
deferred keyword, 280
defined
  assignment, **45**, 78, 82, 91, 110, 147
  operation, 91
  operator, **42**, 44, 47, 78, 82, 110, 146
  variable, 37
definition, 37
definition (of pointer), 79
deleted features, 4, 427
delim= specifier, 217, 221, 326
delimiter, **17**, 18, 196, 217
denormalized number, 224
deprecated features, 399
dereferencing, 49
derived type, **21–24**, 28, 42, 188, 204, 208, 246, 255, 399
  component, 106, 289
  definition, 149, 257
  input/output, 317–320
  literal constant, 22
  parameter enquiry, 256, 259
descriptor, 109
designator, 27, 198
dialect, 2
digits, 169
dim argument, 175
dim procedure, 164
dimension attribute, 23, 100
dimension statement, 402, 409
dimensions of array, 23
direct recursion, 88

direct-access file, **209**, 215–217, 220
`direct=` specifier, 220
disassociation, **37**, 104, 179
disc drive, 190
distinguishable, 270, 371
divide_by_zero, 225
`do concurrent` construct, 359
`do concurrent` statement, 359
`do` construct, **59–62**, 125, 135, 408
`do` construct index, 61
`do while`, 408
dot product, 237
`dot_product`, 174
`double precision` statement, 16, 408
`dp` edit descriptor, 311
`dprod`, 409
`dshiftl`, 379
`dshiftr`, 379
`dt` edit descriptor, 317
dummy
    argument, **74–78**, 92, 93, 100, 144, 163, 405, 406, 421, 424
        allocatable, 105
    procedure, 82
dyadic operator, 33
dynamic type, 268

`e` edit descriptor, 202, 206
EBCDIC standard, 309
edit descriptor, 185, 186, 192, 194, **200–207**, 426
elemental
    assignment, 110
    character function, 166
    function, 163, 164, 234
    mathematical function, 165
    numeric function, 163
    operation, 110
    procedure, 110, **118–119**, 162, 373
    subroutine, 230
`elemental` clause, 118
`else` statement, 56
`elsewhere` statement, 112
`else if` clause, 56
`else if` statement, 56
embedded blanks, 205, 217
`en` edit descriptor, 203, 206
`encoding=` specifier, 310
`end` statement, 63, 68, 73, 190, 419
`end do` statement, 59–407
`end forall` statement, 116
`end function` statement, 70
`end if` statement, 427
`end interface` statement, 81
`end program` statement, 68
`end select` statement, 57, 58
`end` statement, 417
`end subroutine` statement, 70
`end type` statement, 22, 149
`end where` statement, 112, 113
`end=` specifier, 193, 212, 215
endfile record, 193, 209, 215
`endfile` statement, 215, 217
`entry` statement, 424–425
`enum` statement, 253
enumeration, 253
environment variable, 307
`eor=` specifier, **199**, 212
`eoshift`, 178
`epsilon`, 169
`equivalence` statement, 400–402, 404
`erf`, 377
`erfc`, 377
`erfc_scaled`, 377
`err=` specifier, **193, 194**, 212, 214–220
`errmsg=` specifier, 314, 348
error
    message, 314
    recovery, 194
`es` edit descriptor, 203, 206
exception, 193
    flags, 227
    handling, 223
executable statement, 63, 421, 424
`execute_command_line`, 382
`exist=` specifier, 220
existence (of files), 213, 220
`exit` statement, 60, 355
`exp`, 165
explicit interface, **80–82**, 85, 90, 100, 110

explicit-shape array, 153
exponent, 16
  letter, **16**, 202, 203, 408
exponent function, 170
exponentiation, **35**
expression, 24, **33**, 38
extends attribute, 265
extends_type_of, 286
extent, 24, 123
external
  medium, 190
  representation, 208
  subprogram, **67**, 69, 70, 72
external attribute, 151
external statement, **81–82**, 151

f edit descriptor, **202**, 206
field, 185, 419
file, 185
  positioning statements, 214
file= specifier, 216, 219
final statement, 281, 284
final subroutine, 281, 284
finalization, 281–282
findloc, 383
flags (IEEE), 225
floor, 164
flush statement, 324
fmt= specifier, **193**, 194, 212
forall construct, 115, 116, 370
forall statement, 114, 116, 370
form= specifier, 216, 220
format
  specification, **185**, 186, 188, 193, 194, 200, 205, 375
  statement, **190**, 201, 209
formatted
  I/O, **185**, 208
  output, 194
  read, 193
formatted= specifier, 220
Fortran 66, 2
Fortran 77, 3–7, 405, 407, 421, 422
Fortran 90, 3
Fortran 95, 4
fraction, 170
function, **78**, 410, 421
  name, 70, 73, 425
function statement, **95**, 425

g edit descriptor, 201, **204**, 206, 375
gamma, 377
generic
  binding, 317
  identifier, 142
  interface, 90, 371
  name, 82, 87, **90–92**
  type-bound procedure, 277–280
generic statement, 277
get_command, 308
get_command_argument, 308
get_environment_variable, 307
global
  data, 70
  name, 88
go to statement, 63
gradual underflow, 229
graphics display, 192, 194

h edit descriptor, 427
halting, 228
header line, 424
heap storage, 102
hexadecimal constant, 15, 312
High Performance Fortran, 4
high-level language, 2
host, 67, 73
  association, **86**, 149, 413
huge, 169
hypot, 378
hypotenuse function, 238

i edit descriptor, **201**
I/O
  list, 186
  statement, 188
  status statement, 213
  unit, 190
iachar, 166, 309, 313
iall, 379

iand, 172
iany, 379
ibclr, 172
ibits, 172
IBM, 2
ibset, 172
ichar, 166, 313
id= specifier, 321
IEEE
  division, 224
  exceptional value input/output, 323
  flags, 225
  square root, 224
  standard, 169, 223, 224
ieee_arithmetic, 232, 234–236, 387
ieee_class, 234
ieee_class_type, 232
ieee_copy_sign, 234
ieee_datatype, 226
ieee_denormal, 226
ieee_divide, 226
ieee_exceptions, 229–231
ieee_features_type, 225
ieee_flag_type, 229
ieee_get_flag, 230
ieee_get_halting_mode, 230
ieee_get_rounding_mode, 235
ieee_get_status, 231
ieee_get_underflow_mode, 236
ieee_halting, 226
ieee_inexact_flag, 226
ieee_inf, 226
ieee_invalid_flag, 226
ieee_is_finite, 234
ieee_is_nan, 234
ieee_is_negative, 234
ieee_is_normal, 234
ieee_logb, 234
ieee_nan, 226
ieee_next_after, 235
ieee_rem, 235
ieee_rint, 235
ieee_round_type, 232
ieee_rounding, 226
ieee_scalb, 235
ieee_selected_real_kind, 236, 387
ieee_set_flag, 231
ieee_set_halting_mode, 231
ieee_set_rounding_mode, 236
ieee_set_status, 231
ieee_set_underflow_mode, 236
ieee_sqrt, 226
ieee_status_type, 230
ieee_support_datatype, 233
ieee_support_denormal, 233
ieee_support_divide, 233
ieee_support_flag, 230
ieee_support_halting, 230
ieee_support_inf, 233
ieee_support_io, 233
ieee_support_nan, 233
ieee_support_rounding, 233
ieee_support_sqrt, 233
ieee_support_standard, 233
ieee_support_underflow_control, 233
ieee_underflow_flag, 226
ieee_unordered, 235
ieee_value, 235
ieor, 172
if construct, **55–57**, 61
if statement, **56**, 80
image, **333**
image control statement, 337, **348**
image index, **334**
image_index, 351
implicit
  interface, 81
  typing, 133, 412
implicit none statement, 134
implicit statement, 137, 409, **413**, 421
implied-do list, 188
implied-do loop, 88, **126**, 138, 139, 192–193
implied-shape array, 353
import statement, 304–305
impure procedure, **368**
include line, 407
index, 167, 313
indirect recursion, 89
inexact, 225

infinity (signed), 224
inheritance, 265, 267, 279
initial
  line, 419
  point, 214, 217
  value, 137, 358
initialization of components, 141, 358
`inquire` statement, 217, 219–222, 310, 326
inquiry function, 162, 168–170, 176, 230, 232
instruction, 2, 133
`int`, 164
`int16`, 385
`int32`, 385
`int64`, 385
`int8`, 385
integer
  division, 35
  expression, 24, 420
  literal constant, 14
  variable, **20**, 185
`integer` statement, 21, **152**
`integer_kinds`, 386
`intent` attribute, **77–78**, 144, 151, 301
`intent` statement, 144
interface, 80, 330
  block, 43–45, 69, 75, **81–85**, 90, 304, 425
  body, **81**, 82, 85, 92
`interface` statement, **81**, 90
internal
  file, **191–195**, 200, 205, 208
  representation, 185, 200, 208
  subprogram, 67, 73
internationalization, 308
interoperability with C, 243–254
  for coarrays, 343
intrinsic
  assignment, **46**, 110
  data types, 13
  function, 410
  module, 306
  procedure, 161
`intrinsic` attribute, 151
`intrinsic` keyword, 306
`intrinsic` statement, **162**
invalid, 225
`iolength=` specifier, 221
`iomsg=` specifier, 325
`ior`, 173
`iostat=` specifier, **193–194**, 212, 214–216, 218, 220, 306
`iparity`, 379
`is_contiguous`, 362
`is_iostat_end`, 326
`is_iostat_eor`, 326
`ishft`, 173
`ishftc`, 173
ISO/IEC 10646, 309, 310
`iso_c_binding`, 387
`iso_fortran_env`, 306, 385
`iso_c_binding`, 243
iterations, 60

J3, 3, 4, 399

Kanji, 19, 21
keyword
  argument, 83, 85, 92, 289
  call, 161
  for derived type, 257
  specifier, 193
kind
  parameter value, **14**, 16, 19, 20, 37, 41, 93
  type parameter, **13–17**, 21, 36, 40, 135, 163–168, 173, 209, 325
`kind` function, 15–17, 19, 20, **163**
`kind=` specifier, 152

`l` edit descriptor, 203
label, **13**, 63, 85
  scope of, 85
`lbound`, 176, 313
`lcobound`, 351
leading sign, 205
`leadz`, 380
left tab limit, 206
`len`, 168, 313

len= specifier, 152
len_trim, **167**, 313
lexical
  comparison, 167
  token, 10
lge, 167, 383
lgt, 167, 383
line, 11
linked list, 28, 123
list-directed
  I/O, **189**, 214, 217
  input, 195
  output, **190**, 195, 427
literal constant, 13, 14
lle, 167, 383
llt, 167, 383
local entity, 85
lock, **345–347**
log, 165, 314
log10, 165
log_gamma, 377
logical
  array, 126, 175
  assignment, 39
  expression, 39
  literal constant, 19
  variable, 19, 20
logical function, 168
logical statement, 21, **152**
logical_kinds, 386
loop parameter, 59
lower bound, **23**, 99, 149

main program, 67, **68**, 403, 424
make tool, 429
many–one section, 121
mask, 112, 126
mask argument, 176
maskl, 380
maskr, 380
mathematical function, 165
matmul, 174
max, 164, 313
maxexponent, 170
maxloc, 179, 313, 383
maxval, 175, 313
memory leakage, 79, 104, 109, 140, 294
merge, 177
merge_bits, 381
method, 274
MIL-STD 1753, 171
min, 164, 313
minexponent, 170
minimal field width edit descriptor, 202
minloc, 179, 313, 383
minval, 175, 313
mixed-mode expression, 35
mnemonic name, 20
mod, 165
model number, 168, 169, 224
module, 43, 45, 67, **70–72**, 78–82, 142–145, 329–332, 388, 407
  name, 71, 86
  procedure, 90, 330
  subprogram, 67
module procedure statement, 46, **90**, 91, 314
module statement, 71
modulo, 165
monadic operator, 33
move_alloc, 295
MPI, 303
multiplication, 10
  function, 174
mvbits, 173

name, 20, 312
  scope of, 85
name= specifier, 220
named
  constant, **134–137**, 143
  object, 27
named= specifier, 220
namelist
  comments, 198
  data, 197
  group, 143, **155**, 197
  I/O, 197, 327
namelist statement, 155
NaN, 224

NCITS, 3
nearest, 170
nesting, 55, 62, 67, 423
new_line, 324
newunit= specifier, 374
nextrec= specifier, 221
nint, 164
nml= specifier, 197, 212
non-advancing I/O, **198**, 206, 209
non-elemental subroutine, 231, 235
non-numeric types, 14
non_intrinsic keyword, 306
norm2, 378
not, 173
null, 141, 179
null value, 196, 197
nullify statement, 51
num_images, 351
number
  conversion, 185
  representation, 185
number= specifier, 220
numeric
  assignment, 38
  expression, 34
  function, 163, 168
  intrinsic operator, 34
  storage
    association, 400
    unit, 399
  type, 14

o edit descriptor, **202**
object, 27
  coindexed, **336**
object code, 133
object-oriented
  example, 433
object-oriented programming, 265–282
obsolescent features, 4, 419, 422
octal constant, 15, 312
only option, 147
open statement, 190, 209, 210, 213, **216–222**, 310
open statement, 374
operating system, 217
operator, 33, 42–47
  renaming, 297
  token, 42
operator, 91, 92, 297
optional attribute, **83**, 84, 85, 144, 289
optional statement, 144
optional attribute, 389
order of evaluation, 40, 80
order of statements, 68, 70, 71, 73, 134, 190, 420, 425
output list, 186, 192
overflow, 225
overloading, 90
override, 279

p edit descriptor, 206, 324
pack, 177
pad= specifier, 218, 221, 326
parallel processing, 114, 333
parameter attribute, **134**, 137, 151
parameter statement, 409
parameterized derived type, 256, 257
parent data transfer statement, 318
parentheses, 34
parity, 383
pass attribute, 262, 263
passed-object dummy argument, 263
pause statement, 427
pending= specifier, 321
percent, 22
pointer, 27, 37, **49–50**, 60, 75–79, 82, 103–104, 123, 124, 140, 149, 187, 208, 269
  allocation, 28, 103
  argument, 75, 388
  assignment, 42, **48**, 50, 79, 115, 270, 296
    statement, **49**, 124
  associated, **37**, 103
  association, **37**, 75, 125, 301
  component
    of coarray, 341–342
  disassociated, **37**, 104, 140
  expression, 48

function, 79, 104, 366
initialization, 140
intent, 301
undefined, **37**, 140
pointer attribute, **27**, 49, 75, 78, 100, 123, 144, 149, 151, 399
pointer statement, 144
polymorphic entity, 267–269, 292
popcnt, 380
poppar, 380
pos= specifier, 323
position= specifier, 217, 221
positional argument, 83, 92
precedence of operators, 43
precision, 16, 17
precision function, 17, 170
preconnection (of files), 213, 307
present, 84, 163
print statement, 193, **194**, 215, 217
private attribute, 142, **143**, 289, 315, 404
private statement, **142**, 150
procedure, 67
argument, 82, 370
pointer, 261
component, 261, 262
variable, 261
procedure statement, 259–261, 276, 314
processor dependence, 7
product, 175
program, 10, 67
name, 68
termination, 348–349
unit, 10, **67**, 70, 74, 216, 402, 403, 405, 421
program statement, 68
protected attribute, 296, 297
prototype, 249
public attribute, **142**, 143, 289
public statement, **142**, 150
pure procedure, **117–118**, 153

radix, 170
random-access file, 209
random_number, 181
random_seed, 181
range, 14–16, 61
range, 15–17, **170**
rank, 23–25, 28, 110, 356
read statement, 186, **193**, 192–194, 196–199, 209, 211, 214, 215, 217, 326
read= specifier, 221
readwrite= specifier, 221
real
literal constant, 15, 16
operand, 36
variable, 20, 186
real function, 164
real statement, 21, **152**
real-time clock, 180
real128, 385
real32, 385
real64, 385
real_kinds, 386
reallocation, 108, 294, 373
rec= specifier, 209, 212
recl= specifier, 217, 220, 222, 306
record, **185**, 193, 195, 198–209, 213–215
length, 217, 220
recursion, **88–89**, 145, 425
recursive input/output, 324, 374
register, 2
relational expression, 39
repeat, 168
repeat count, **138**, 186, 196, 200, 201, 375
reserved words, 20
reshape, 177
result clause, **88**, 425
return statement, **77**, 424
reversion, 201
rewind statement, 214
round= specifier, 325
rounding, 228
rounding modes, 224
rrspacing, 171

s edit descriptor, 206
safety, 3, 4
same_type_as, 287

`save` attribute, 137, **145**, 404
`save` statement, 145
`save` attribute, 388
`scale`, 171
scale factor, 206
`scan`, 167, 313
scope, **85**, 145
scoping unit, **85**, 186, 190
segment, **343–344**
  unordered, 344
`select case` statement, 57
`select type` construct, 268, 270, **272**, 274
`selected_char_kind`, 309
`selected_int_kind`, 14, **171**, 356
`selected_real_kind`, 16, **171**, 384
selector, 58
semantics, 7
semicolon, 12
separator, 10, 196
`sequence` attribute, **399**, 401, 403
`sequence` statement, 399
sequence type, 399
sequential file, **209**, 210, 213, 217, 220
`sequential=` specifier, 220
`set_exponent`, 171
shallow copying, **108**, 373
shape, **24**, 42, 46, 48, 110, 134, 176, 405
`shape` function, 176, 313
shell script, 429
`shifta`, 381
`shiftl`, 381
`shiftr`, 381
side-effect, 79, 117
`sign`, 165, 169
`sign=` specifier, 325
significance, 16
  of blanks, 11
simply contiguous, **364**
`sin`, 166, 376
`sinh`, 166, 376
`size`, 176, 313
size (of array), **23**, 176
`size=` specifier, 199, 212, 326
slash edit descriptor, **207**, 209
source
  code, 2, 133, 429
  form, 3, **11**, 18, 419
`source=` clause, 291, 293
`sp` edit descriptor, 206
spaces, 207
`spacing`, 171
specific name, 82, **90**, 410
specific type-bound procedure, 274, 279
specification
  expression, 153, **153**, 298
  function, 153
  statement, **68**, 133, 421
`spread`, 178
`sqrt`, 166, 314
square brackets, 55, 298, 333
`ss` edit descriptor, 206
stack, 101
`stat=` specifier, 103, 104, 348
statement, **10–12**, 33, 312, 419
  function, 421
  label, **13**, 186, 190, 193, 407, 424
  separator, 12
`status=` specifier, 216, 218
`stop` statement, 69, 355
stop code, 69, 349, 355
storage, 104
  allocation, 2, 103, 104
  association, 3, 4, **399**
  system, 213
  unit, 401
`storage_size`, 384
stream access input/output, 323
`stream=` specifier, 326
stride, 122
string-handling
  function, 167
  inquiry function, 168
  transformational function, 168
strong typing, 133
structure, **22**, 27, 135
  component, 123
  constructor, **22**, 42, 138, 289, 354
  of pointers, 123
submodule, 329–332

entity, 331
of submodule, 331
procedure, 331
`submodule` statement, 330, 331
subobject, 27, 120–123
subprogram, **67**
subroutine, **67**, 145, 423
name, **70**, 73
`subroutine` statement, 70, **95**, 425
subscript, 23–26, 29, **120**, 125, 271
substring, 8, **26**, 120, 123
`sum`, 175
`sync all` statement, 337
`sync images`, **344–345**
`sync memory`, **347**, **413–415**
synchronization, 302, **337**, **343–348**
syntax, 9, 20, 33
rules, 7
`system_clock`, 180, 313

`t` edit descriptor, 206
tabulation, 206
`tan`, 166, 376
`tanh`, 166, 376
target, 28, 48, 50, 103, 104
`target` attribute, **50**, 76, 144, 151, 403, 406
`target` statement, 144
`target` attribute, 388
terminal point, 214, 217
termination, 348–349
`this_image`, 351
time, 180
`tiny`, 170
`tl` edit descriptor, 206
token, 10, 12, 13
`tr` edit descriptor, 206
`trailz`, 380
`transfer`, 174
transfer of allocation, 295
transformational function, 162, 236
`transpose`, 179
`trim`, 168
type, 133
allocation, 291, 292
conversion, 163
declaration statement, 20–22, 150
extensible, 267
extension, 265–267, 284
name, 69
parameter, 21, 150, 255, 256, 291
enquiry, 256, 259
specification, 152
specification, 152
statement, 150
`type` statement (see also derived type), 22, 69, 152
`type is` statement, 272
`type is` guard, 273
type-bound procedure, 274–280, 354

`ubound`, 176, 313
`ucobound`, 351
ultimate component, 187
unary
operation, 46
operator, **33**, 39, 43
undefined variable, **37**, 145
underflow, 225, 229
underscore, **9**, 19, 20
unformatted I/O, **208**, 219, 221
`unformatted=` specifier, 220
Unicode, 309
unit, 190
number, **190**, 193, 213, 215, 216, 219, 307
`unit=` specifier, **193**, 194, 212, 214–216, 218, 219
unix, 308
unlimited polymorphic
entity, 274, 292
pointer, 269
`unlock`, **345–347**
`unpack`, 177
unspecified storage unit, 399
upper bound, **23**, 99, 149
`use` statement, 72, 86, **146**, 305
use association, **86**, 149, 413
UTF-8 format, 310

`value` attribute, 248

variable, 13, **20**, 27, 186
    (defined), **37**, 153
    (undefined), **37**, 145, 153
vector subscript, 25, 121, **121**
`verify`, 168, 313
`volatile` attribute, 301–304
    coarray, 343
`volatile` statement, 301

`wait` statement, 321, 322
WG5, 3–5, 399
`where` construct, 111, 112, 115
`where` statement, 111, 126
`while`, 408
whole coarray, **335**
`write` statement, 194, 197–200, 211, 215, 217, 326
`write=` specifier, 221

`x` edit descriptor, 206, 207
X3J3, 3, 4, 399

`z` edit descriptor, 202
zero (signed), 224
zero-length string, **18**, 26, 39
zero-sized array, **99**, 126