Fortran

Numbered references are to the currently valid Fortran 2018 standard (ISO/IEC 1539-1:2018); they are often entry points to further cross-references inside the standard. Otherwise, processor/platform dependencies and indications of best practices are pointed out where appropriate.

Beginners' content

- Concept of program units (5.2.1-4, 6.3.1): program (14.1) and module (14.2, simplest usage).
- Layout of tokens and program lines (6.2) in free source form (6.3.2).
- Concept of execution sequence (5.3.1-5) for a single image, program termination via END PROGRAM or STOP (5.3.7).
- Type system (7.1):
  - Intrinsic types (7.4) and type parameters (7.2) for intrinsic types
  - Simple derived types (7.5)
  - Implicit typing (8.7), specifically its avoidance (best practice)
- Declaration and processing of data (scalar or array, see below) of intrinsic or simple derived type:
  - Declaration of objects (8.2) for non-dynamic entities
  - Intrinsic operations (5.5.6, 7.4.2)
  - Expressions (10.1) and their evaluation; conversions and recommendations on their use (best practice)
  - Assignment (10.2) for intrinsic and simple derived types
- Concept of attribute (8.1, 8.5), specifically the DIMENSION (8.5.8) attribute (for arrays, see below) and PARAMETER (8.5.13) attribute (for constants).
- Array concept (5.4.6):
  - Arrays (9.5) of intrinsic or simple derived type; rank, shape and bounds of an array (5.4.6)
  - Construction of array values (7.8), implied-do loops
  - Array sections, array element sequence (9.5.3) and performance impact of their use (best practice)
- Execution control via block constructs (11.1), specifically BLOCK (11.1.4), DO (11.1.7), IF (11.1.8) and SELECT CASE (11.1.9)
- Intrinsic procedures (16.9):
  - Overview of simple and commonly used (elemental) mathematical and string processing intrinsic functions
  - Conversion intrinsics
  - Inquiry intrinsics for array properties
- Fortran environment:
  - The ISO_FORTRAN_ENV intrinsic module (16.10.2) with names for commonly used intrinsic type parameter values, for standard storage size values, and for preconnected I/O unit values.
  - Intrinsic procedures (16.9) for processing environment information (variable values and command line arguments)
- Procedures (15):
  - Function vs. subroutine (15.2.1); concept of dummy argument (15.6.2) for static objects of intrinsic or simple derived type; function results and the RESULT clause (15.6.2.2)
  - Invoking a procedure (15.5.1); association between actual and dummy argument (15.5.2) by position or keyword; also includes rules for aliasing and side effects and recommendations (best practice)
  - Procedure interfaces (15.4): prefer explicit to implicit interfaces (15.4.2, best practice), use interface blocks (15.4.3.2) where necessary
  - Module procedures and internal procedures (15.2.2.2, best practice)
  - (Non-)recursive (15.6.2) and PURE (15.7) procedures
  - Passing arrays to procedures: assumed-size (8.5.8.5), explicit-shape (8.5.8.2) and assumed-shape (8.5.8.3) dummy arguments
  - The INTENT (8.5.10, best practice), OPTIONAL (8.5.12), and VALUE (8.5.18) attributes
- I/O facilities (12, 13):
  - Concept of external file (12.3.1) and record (12.2.1)
  - Connecting a unit (12.5.3) with a file (12.5.4); the OPEN (12.5.6) and CLOSE (12.5.7) statements for sequential (12.3.3.2) formatted (12.2.2) files
  - Data transfer I/O lists with implied-do loops (12.6.3)
  - Use of preconnected units (12.5.5, 12.5.1)
  - READ and WRITE (or PRINT) statements (12.6) for list-directed (13.10) or format-controlled (13.1-4) I/O data transfers
  - The most commonly used data edit descriptors (13.7.1-4) and control edit descriptors (13.8); character string editing (13.9)
- Interoperability with C (18):
  - Type parameter values (18.2.2) supplied by the intrinsic module ISO_C_BINDING (18.2)
  - Interoperation of scalar or arrays of intrinsic C type (18.3.1) or simple C struct type (18.3.3) through BIND(C) procedure calls (18.3.6, 18.10.2)
  - The C_SIZEOF procedure (18.2.3)

Intermediate content

- Intrinsic procedures (16.9):
  - Classification according to inquiry and transformational procedures (16.1)
  - Treatment of array arguments, including semantics of MASK and DIM (16.2), in particular for reduction functions
  - Numerical models (16.4) and their associated intrinsics
- Modules (14.2):
  - General syntax and semantics (14.2.1)
  - Use association (14.2.2) and accessibility (8.5.2); the PROTECTED (8.5.15) attribute
  - Limited namespace management via USE, ONLY and renaming of module entities (best practice)
  -Opaque (7.5.4.1) and private types (best practices)
- Constant expressions (10.1.12), initializations (8.4) and specification expressions (10.1.11)
  - DATA statement (8.6.7) and its implied-do semantics
  - Host association (19.5.1.4) and scoping rules (19.1)
  - Controlling host access via the IMPORT statement (8.8)
- Dynamic data:
  - The ALLOCATABLE (8.5.3), POINTER (8.5.14) and TARGET (8.5.17) attributes
Dynamic association (9.7) and lifecycle management of dynamic objects (semantics of allocation, deallocation and pointer association)
Auto-allocation on assignment (10.2.1.3), moving an allocation (16.9.137), checking allocation status (16.9.11) and association status (16.9.16)
Allocatable scalars and deferred-length strings
Pointer assignment (10.2.2), including rules for rank changing and bounds remapping
Potential memory management issues for POINTER objects (implementation dependencies and best practices)
Automatic objects (8.3); discussion of heap vs. stack allocation (best practices)

Array processing (9.5):
Passing arrays to procedures: deferred-shape (8.5.8.4) and assumed-rank (8.5.8.7) dummy arguments
The WHERE (10.2.3) array assignment construct
The SELECT RANK (11.1.10) block construct
The CONTIGUOUS attribute (8.5.7) and its semantics; simply contiguous array designators (9.5.4). Discussion of performance tradeoffs (best practice)

Procedures:
Elemental procedures (15.8)
Factory procedures: subroutines with dynamic dummy arguments (15.5.2.5-7); functions with dynamic result variables (15.6.2.2)
Using procedure arguments (15.5.2.9) for a functional programming style
IMPURE procedures (15.6.2.1, 15.7)
Procedure pointers (15.2.2.4, 8.5.14)

Object-based programming:
Type components with the POINTER or ALLOCATABLE attributes (7.5.4.1)
Semantics of such type components for structure construction (7.5.10), assignment (10.2.1), and in procedure invocations (8.5.10)
Avoiding resource leaks through final procedures (7.5.6)
The ASSOCIATE block construct (11.1.3)
Derived types parameterized by length and kind components (7.2, 7.5.3); simple scenarios for use of assumed and deferred length type parameters. Using parameterized types for performance-driven design, based on memory layout (best practice).

Generic programming:
Named interfaces (15.4.3), distinguishability of specific procedures with the same generic identifier (15.4.3.4.5), and resolution of generic calls at compilation time (15.5.5.1)
Defined operations (10.1.6) and assignment (10.2.1.4)

I/O facilities (12, 13):
File inquiry (12.10)
File positioning (12.3.4) and non-advancing I/O (12.6.2.4); file positioning statements (12.8)
Concept of file storage unit (12.3.5)
Further file access modes: direct access (12.3.3.3), stream access (12.3.3.4)
Unformatted I/O (12.2.3); tradeoffs of performance vs. portability (best practice)
Internal files (12.4)
Handling groups of key-value pairs via namelist I/O (13.11)
I/O error handling (12.11)

Interoperability with C (18):
Interoperation of global data (18.9)
Interoperation with C pointer types (18.3.2); the C_LOC, C_ASSOCIATED, and C_F_POINTER module procedures (18.2.3)
Interoperation with C procedure pointers (18.3.2); the C_FUNLOC and C_F_PROCPOINTER module procedures (18.2.3)

Advanced content

IEEE Arithmetic and floating point exception handling (17)
The IEEE.EXCEPTIONS, IEEE_ARITHMETIC, and IEEE_FEATURES intrinsic modules (17.1) and the types defined therein (17.2)
IEEE conforming floating point representations and associated real kind numbers; the procedures IEEE_SELECTED_REAL_KIND (17.11.34) and IEEE_SUPPORT_DATATYPE (17.11.48)
IEEE exceptions (17.3) and associated flags; signaling vs. non-signaling exceptions
Rounding (17.4) and its control (17.11)
Underflow mode (17.5) and its control (17.11)
Halting (17.6) and its control (17.11)
Managing (17.11) the floating point status (17.7)
Exceptional floating point numbers (17.8) and their creation (17.11)

Object-oriented programming:
Type extension, abstract types and inheritance (7.5.7)
Data and interface polymorphism - declaration of variables with the CLASS specifier (7.3.2.1, 7.3.2.3); dynamic and declared type; type compatibility
Typed, sourced and molded allocation (9.7.1)
Type-bound procedures (7.5.5) and object-bound procedures (10.2.2.2); the PASS and NOPASS attributes (7.5.4.5)
Overriding type-bound procedures (7.5.7.3) and the NON_OVERRIDABLE attribute (7.5.5)
Run time type and class identification with the SELECT TYPE block construct (11.1.11); type inquiry intrinsics SAME_TYPE_AS (16.9.165) and EXTENDS_TYPE_OF (16.9.76).
Abstract interfaces (15.4.3.2) and deferred type-bound procedures (7.5.5)
Unlimited polymorphic objects (7.3.2.3)
Polymorphic assignment (10.2), overloading of the structure constructor (15.4.3.4) and polymorphic object construction
Generic type-bound procedures and operations (7.5.5)
Invocation of specific and generic type-bound procedures (15.5.6)

Dependency inversion with submodules (5.2.5):
Syntax and semantics of submodules (14.2.3)
Separate module procedures (15.6.2.5)
Avoidance of compilation cascades, implementation of dependency-inversed object-oriented patterns (use cases, best practices)

Parallel programming:
SPMD-style multi-image execution (5.3.4-5)
Coarray concept (5.4.7): The CODIMENSION attribute (8.5.6) and symmetric data decomposition
Coarray use: programming one-sided (put- or get-style) data exchange between images through coindexing (9.6)
• Intrinsic procedures for parallel execution control and coarray inquiry: NUM_IMAGES (16.9.145), THIS_IMAGE (16.9.190), IMAGE_INDEX (16.9.97)
• Synchronization: Concept of an execution segment (11.6.2) and the need for image control statements (11.6.1) to impose segment ordering; collective synchronization with SYNC ALL (11.6.3)
• Collective intrinsic subroutines (16.6) for data redistribution and computation: CO_REDUCE (16.9.49), CO_MIN (16.9.48), CO_MAX (16.9.47), CO_SUM (16.9.50), CO_BROADCAST (16.9.46)
• One-sided segment ordering through EVENT POST (11.6.7) and EVENT WAIT (11.6.8)
• Mutual exclusion via CRITICAL blocks (11.1.6) or locks (11.6.10); synchronization of image subsets via SYNC IMAGES (11.6.4)
• Unsynchronized coarray updates through atomic procedures (16.5, 16.9.20-30); support for user-defined segment ordering via SYNC MEMORY (11.6.5)
• Coarray dummy arguments (15.5.2.8,13)
• Dynamic coarrays: symmetric and unsymmetric allocation (9.7.1), the MOVE_ALLOC intrinsic subroutine (16.9.137) for coarrays
• Features and limitations of coarray use in an object-based or object-oriented context (allocatable/polymorphic coarrays or coarray components)
• Creation and execution of (disjoint) image teams via the FORM TEAM statement (11.6.9) and the CHANGE TEAM block construct (11.1.5), respectively
• Cross-image access (9.6) to established coarrays (5.4.8) within teams and across team boundaries
• Team-wide synchronization with the SYNC TEAM image control statement (11.6.6)

I/O facilities (12, 13):
• Asynchronous data transfers (12.6.2.5) and the ASYNCHRONOUS attribute (8.5.4); assuring completion of asynchronous data transfers (12.7)
• User-defined derived type I/O: Derived type edit descriptor (13.7.6) and its type-bound or generic procedures (12.6.4.8)

Community-targeted and domain-specific content

Interoperability topics (18) for advanced library interface development
• Binding non-interoperable Fortran interfaces to C through use of a C descriptor (18.3.6, 18.4)
• Manipulation of a C descriptor (18.5) and rules for its use (18.6-8)
• Assumed-type dummy argument (7.3.2) and its interoperability semantics
• Using assumed-type and assumed-rank entities for effective suppression of procedure argument TKR checking
• Extended use of the ASYNCHRONOUS attribute (18.10.4)

Modernization of legacy code
• Compiler support for flagging non-standard, standard-level, obsolescent, or removed features (tools)
• Fixed source form (6.3.3, B.3.7) and conversion tools
• Non-standard notations for intrinsic types and type promotion by the compiler
• CHARACTER* declaration (7.4.4.2, B.3.8)
• Legacy notation for operators (10.1.5)
• Legacy execution control:
  • Branching (11.2)
  • arithmetic IF (deleted)
  • computed GOTO (11.2.3)
  • assigned GOTO and ASSIGN (deleted)
  • non-block DO loop (deleted) and labeled DO loop (B.3.10)
  • non-integer loop control variable (deleted)
• Legacy type concepts: SEQUENCE types (7.5.2.3) and (non-standard) record types
• Procedures:
  • Implicit interfaces (15.4.2, 15.4.3.8) and external procedures
  • Arguments declared without INTENT (B.3.10)
  • Statement functions (15.6.4, B.3.4)
  • Alternate return arguments (15.6.2.7, B.3.2)
  • Assumed character length function result (B.3.6)
  • ENTRY statement (B.3.9)
• Specific names for intrinsic functions (B.3.12)
• COMMON blocks and their initialization with BLOCK DATA (B.3.11)
• Enforcing storage association with EQUIVALENCE (B.3.11); replacement by appropriate POINTER entities, ALLOCATABLE entities, or the TRANSFER intrinsic subroutine (16.9.193)
• Non-standard dynamic memory with Cray Pointers and its replacement by either C interoperability features or dynamic Fortran objects
• I/O
  • Hollerith edit descriptor (deleted)
  • vertical format control (deleted)
  • PAUSE statement (deleted)
• Array assignments with FORALL (B.3.13)

Design of user-defined low-level data representations
• The TRANSFER intrinsic subroutine (16.9.193)
• Bit model and sequences (16.3) and intrinsic procedures for bit manipulation
• BOZ literals (7.7) and their use

Resilience for large-scale parallel programs
This covers the concept of continuing program execution in the face of image failure. A processor is not obliged to support this feature.
• Concept of failed image (5.3.6) and the STAT FAILED IMAGE constant
• Termination model for Fortran (5.3.7) and forcing an image failure via FAIL IMAGE (11.5)
• Explicit status query on image control statements (9.7.4, 11.6.11) and collective procedures (16.6) as a prerequisite for resilient applications
• Semantic differences between stopped and failed images (best practices)
• Effect of coindexed accesses involving failed images (9.6)
• Use of teams to maintain program execution integrity (C.6.8)