Clause	Advice		
6.2	Type system		
		Use kind values based on the needed range for integer types via the `selected_int_kind` intrinsic procedure and based on the range and precision needed for real and complex types via the `selected_real_kind` intrinsic procedure.	
		Use explicit conversion intrinsics for conversions of values of intrinsic types, even when the conversion is within one type and is only a change of kind. Doing so alerts the maintenance programmer to the fact of the conversion, and that it is intentional.	
		Use inquiry intrinsic procedures to learn the limits of a variable's representation and thereby take care to avoid exceeding those limits.	
		Use derived types to avoid implicit conversions.	
		Use compiler options when available to detect during execution when a significant loss of information occurs.	
		Use compiler options when available to detect during execution when an integer value overflows.	
6.3	Bit representation		
		Use the language-provided intrinsics whenever bit manipulations are necessary, especially those that occupy more than one integer.	
		Use the intrinsic procedure `bit_size` to determine the size of the bit model supported by the kind of integer in use.	
		Be aware that the Fortran standard uses the term "left-most" to refer to the highest-order bit, and the term "left" to mean towards the highest-order bit (as in `shiftl`).	
		Do not use compiler extensions that allow variables of logical type to hold bit string values, because the results may vary between implementations.	
		Avoid compiler extensions that accept BOZ constants in non-standard usage.	
		Encapsulate bit strings inside derived types to exclude numeric operations on them.	
6.4	Floating Point Arithmetic		
		Use procedures from a trusted library to perform calculations where floating-point accuracy is needed. Understand the use of the library procedures and test the diagnostic status values returned to ensure the calculation proceeds as expected.	

		Avoid creating a logical value from a test for equality or inequality between two floating-point expressions. Use compiler options where available to detect such usage.	
		Do not use floating-point variables as loop indices, a deleted feature; use integer variables instead. A floating-point value can be computed from the integer loop variable as needed.	
		Use intrinsic inquiry procedures to determine the limits of the representation in use when needed.	
		Avoid the use of bit operations to get or to set the parts of a floating point quantity. Use intrinsic procedures to provide the functionality when needed.	
		Use the intrinsic module procedures to determine the limits of the processor's conformance to IEEE 754, and to determine the limits of the representation in use, where the IEEE intrinsic modules and the IEEE real kinds are in use.	
		Use the intrinsic module procedures to detect and control the available rounding modes and exception flags, where the IEEE intrinsic modules are in use.	
6.5	Enumerator issues		
		Use enumeration values in Fortran only when interoperating with C procedures that have enumerations as formal parameters and/or return enumeration values as function results.	
		Ensure the interoperability of the C and Fortran definitions of every enum type used.	
		Ensure that the correct companion processor has been identified, including any companion processor options that affect enum definitions.	
		Do not use variables assigned enumeration values in arithmetic operations, or to receive the results of arithmetic operations if subsequent use will be as an enumerator.	
6.6	Conversion errors		
		Use the kind selection intrinsic procedures to select sizes of variables supporting the required operations and values.	
		Use a temporary variable with a large range to read a value from an untrusted source so that the value can be checked against the limits provided by the inquiry intrinsics for the type and kind of the variable to be used.	

		Use a temporary variable with a large range to hold the value of an expression before assigning it to a variable of a type and kind that has a smaller numeric range to ensure that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.	
		Use derived types and put checks in the applicable defined assignment procedures.	
		Use static analysis or compiler features to identify conversions that can lose or corrupt information.	
		Use compiler options when available to detect and report during execution when a loss or corruption of information occurs.	
		Consider using simple derived types to hold numeric values that can represent different unit systems (such as radians vs degrees) and provide explicit conversion functions as needed.	
		Include an IOSTAT variable in each IO statement and check its value to ensure no errors occurred.	
6.7	String termination (none)		
6.8	Buffer boundary violation (overflow)		
		Ensure that consistent bounds information about each array is available throughout a program.	
		Enable bounds checking throughout development of a code. Disable bounds checking during production runs only for program units that are critical for performance.	
		Use whole array assignment, operations, and bounds inquiry intrinsics where possible.	
		Obtain array bounds from array inquiry intrinsic procedures wherever needed. Use explicit interfaces and assumed-shape arrays to ensure that array shape information is passed to all procedures where needed, and can be used to dimension local arrays.	
		Use allocatable arrays where array operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.	
		Use allocatable character variables where assignment of strings of varying sizes is expected so the left-hand side character variable is re-allocated as needed.	
		Use intrinsic assignment for the whole character variable rather than looping over substrings to assign data to statically-sized character variables so the truncate-or-blank-fill seman-tic protects against storing outside the assigned variable.	

		Consider using the `iostat=` specifier when there is a risk that an internal file is too small for the output sent to it.	
6.9	Unchecked array indexing		
		Ensure that consistent bounds information about each array is available throughout a program.	
		Enable bounds checking, when available, throughout development of a code. Disable bounds checking during production runs only for program units that are critical for performance.	
		Use whole array assignment, operations, and bounds inquiry intrinsics where possible.	
		Obtain array bounds from array inquiry intrinsic procedures wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable arrays as procedure dummy arguments to ensure that array shape information is passed to all procedures where needed, and can be used to dimension local arrays.	
		Use allocatable arrays where array operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.	
		Declare the lower bound of each array extent to fit the problem, thus minimizing the use of subscript arithmetic.	
6.10	Unchecked array copying - no specific guidance		
6.11	Pointer type conversions		
		Avoid implicit interfaces; use explicit interfaces instead.	
		Avoid the use of C-style pointers, unless necessary to interface with C programs.	
		Avoid sequence types.	
6.12	Pointer arithmetic - no specific guidance		
6.13	Null pointer dereference		
		Ensure that all pointers have a defined association status before use, either by initialization or by pointer assignment.	
		Consider using `allocatable` instead of `pointer` when possible, since the allocation status of allocatable variables or allocatable components cannot be undefined.	
		Use static analysis tools and compiler options where available to enable pointer checking during development of a code.	

		Use the `associated` intrinsic procedure before referencing a target through a pointer if there is any possibility of the pointer being disassociated.	
		Use default initialization in the declarations of pointer components.	
6.14	Dangling reference to heap		
		Use allocatable objects in preference to pointer objects whenever the facilities of allocatable objects are sufficient.	
		Use compiler options where available to detect dangling references.	
		Use compiler options where available to enable pointer checking throughout development of a code. Disable pointer checking during production runs only for program units that are critical for performance.	
		Do not pointer-assign a pointer to a target if the pointer might have a longer lifetime than the target or the target attribute of the target. Check actual arguments that are argument associated with dummy arguments that are given the `target` attribute within the referenced procedure.	
		Check for successful deallocation when deallocating a pointer by using the `stat=` specifier.	
6.15	Arithmetic wrap-around error		
		Follow the guidance of ISO/IEC 24772-1:2019 clause 6.15.5	
		Use the intrinsic procedure `selected_int_kind` to select an integer kind value that will be adequate for all anticipated needs.	
		Use compiler options where available to detect during execution when an integer value overflows.	
6.16	Using shift operators for multiplication and division		
		Do not use shift intrinsics where integer multiplication or division is intended.	
6.17	Choice of clear names		
		Declare all variables and use `implicit none` to enforce this.	
		Do not use consecutive underscores in a name.	
		Do not use keywords as names when there is any possibility of confusion.	
		Be aware of language rules associated with the case of external names and with the attribute `bind(C)`.	

6.18	Dead Store - no specific guidance		
6.19	Unused variable - no specific guidance		
6.20	Identifier name reuse		
		Do not reuse a name within a nested scope.	
		Clearly comment the distinction between similarly-named variables, wherever they occur in nested scopes.	
		Be aware of the scoping rules for statement entities and construct entities	
6.21	Namespace issues		
		Avoid implicit typing; always declare all variables; and use `implicit none` to enforce this.	
		Use a global `private` statement in all modules to require explicit specification of the `public` attribute.	
		Use an `only` clause on every `use` statement.	
		Use renaming to resolve name collisions.	
6.22	Missing initialization of variables		
		Favour explicit initialization in executable statements for objects of intrinsic type and default initialization for objects of derived type. When providing default initialization, provide	
		Use type value constructors to provide values for all components.	
		Use compiler options, where available, to find instances of use of uninitialized variables.	
		Use other tools, for example, a debugger or flow analyzer, to detect instances of the use of uninitialized variables.	
6.23	Operator precedence and associativity		
		Consult the Fortran reference manual or suitable textbooks for definitive information on specific operator precedence and associativity issues	
		Is there a difference in IEEE 754 precedence and associativity?	
6.24	Side effects and order of evaluation		
		Replace any function with a side effect by a subroutine so that its place in the sequence of computation is certain.	
		Assign function values to temporary variables and use the temporary variables in the original expression.	
		Declare a function as `pure` whenever possible.	

6.25	Likely incorrect expression		
		Use an automatic tool to simplify expressions.	
		Check for assignment versus pointer assignment carefully when assigning to names having the pointer attribute.	
		Enable the compiler's detection of nonconforming code.	
6.26	Dead and deactivated code		
		Use an editor or other tool that can transform a block of code to comments to do so with dead or deactivated code.	
		Use a version control tool to maintain older versions of code when needed to preserve development history.	
6.27	Switch statements and static analysis		
		Cover cases that are expected never to occur with a case default clause to ensure that unexpected cases are detected and processed, perhaps emitting an error message.	
6.28	Demarcation of control flow		
		Use the block form of the do-loop, together with cycle and exit state-ments, rather than the non-block do-loop.	
		Use the `if` construct or `select case` construct whenever possible, rather than statements that rely on labels, that is, the arithmetic `if` and `go to` statements.	
		Use names on block constructs to provide matching of initial statement and end statement for each construct.	
6.29	Loop control variable abuse		
		Ensure that the value of the iteration variable is not changed other than by the loop control mechanism during the execution of a `do` loop.	
		Verify that where the iteration variable is an actual argument, it is associated with an `intent(in)` or a `value` dummy argument.	
6.30	Off-by-one error		
		Declare array bounds to fit the natural bounds of the problem.	
		Declare interoperable (with C) arrays with the lower bound 0.	
		Use `lbound` and `ubound` intrinsics to specify loop bounds instead of numeric literals.	
6.31	Unstructures programming		

		Use the compiler or static analysis tools to detect unstructured programming and the use of old or obsolescent features.	
		Use a tool to automatically refactor unstructured code.	
		Replace unstructured code manually with modern structured alternatives only where automatic tools are unable to do so.	
		Use the compiler or other code analysis tool to detect archaic usage.	
6.32	Passing parameters and return values		
		Specify explicit interfaces by placing procedures in modules where the procedure is to be used in more than one scope, or by using internal procedures where the procedure is to be used in one scope only.	
		Specify argument intents to allow further checking of argument usage.	
		Specify `pure` (or `elemental`) for procedures where possible for greater clarity of the programmer's intentions.	
		Use a compiler or other tools to automatically create explicit interfaces for external procedures.	
6.33	Dangling references to stack frames		
		Do not pointer-assign a pointer to a target if the pointer association might have a longer lifetime than the target or the `target` attribute of the target.	
		Use allocatable variables in preference to pointers wherever they provide sufficient functionality.	
6.34	Subprogram signature mismatch		
		Use explicit interfaces, preferably by placing procedures inside a module or another procedure.	
		Use a processor or a static analysis tool that check all interfaces, especially if this can be checked during compilation with no execution overhead.	
		Use a processor or other tool to create explicit interface bodies for external procedures.	
6.35	Recursion		
		Prefer iteration to recursion, unless it can be proved that the depth of recursion can never be large.	
6.36	Ignored error status and unhandled exception		

		When the default behaviour of program termination is undesirable, code a status variable for all statements that support one, examine its value prior to continuing execution for faults that cause termination, and take appropriate action.	
		Check and respond to all status values that might be returned by an intrinsic procedure or by a library procedure.	
6.37	Type-breaking reinterpretation of data		
		Do not use `common` to share data. Use module variables instead.	
		Do not use `equivalence`. If the intent is to save storage space, use allocatable data instead.	
		Do not use `entry`. Use a module containing any private data items, with a module procedure for each entry point and the shared code in a private module procedure.	
		Use compiler options where available to detect violation of the rules for `common`, `equivalence`, and `entry`.	
6.38	Deep vs. shallow copying		
		Use allocatable components in preference to pointer components.	
		Copy the objects referred to by pointer components if there is any possibility that the aliasing of a shallow copy would affect the application adversely.	
		When possible, use 1 as the lower bound of array dimensions to avoid indexing mistakes.	
6.39	Memory leaks and heap fragmentation		
		Use allocatable data items rather than pointer data items whenever possible.	
		Use final routines to free memory resources allocated to a data item of derived type.	
		Use a tool during testing to detect memory leaks.	
6.40	Templates and generics - N/A		
6.41	Inheritance		
		Declare a type-bound procedure to be `non overridable` when neces-sary to ensure that it is not overridden by subclasses.	
6.42	s of the Liskov substitution principle - no additional guidance		
6.43	Redispatching		
		Where redispatching is undesirable, do not prefix the call of a procedure with the name of an object.	
6.44	Polymorphic variables		

		Ensure that the default case in `select type` statements is handled.	
6.45	Extra intrinics		
		Specify that an intrinsic or external procedure has the `intrinsic` or `external` attribute, respectively, in the scope where the reference occurs.	
		Use compiler options to detect use of non-standard intrinsic procedures.	
6.46	Argument passing to library functions		
		Use libraries from reputable sources with reliable documentation and understand the documentation to appreciate the range of acceptable input.	
		Verify arguments to library procedures when their validity is in doubt.	
		Use condition constructs such as `if` and `where` to prevent invocation of a library procedure with invalid arguments.	
		Provide explicit interfaces for library procedures. If the library provides a module containing interface bodies, use the module.	
6.47	Inter-language calling		
		Correctly identify the companion processor, including any options affecting its types.	
		Use the C interoperability features of Fortran (the `iso_c_binding` module, the `ISO_Fortran_binding.h` header file, and the `bind(C)` attribute), and use the correct constants therein to specify the type kind values needed.	
		Use the value attribute as needed for dummy arguments.	
		Perform IO on any given file in one programming language only; consider restricting all IO to one language system only.	
6.48	ynamically-linked code and self-modifying code - N/A		
6.49	Library signature		
		Use explicit interfaces for the library code if they are available. Avoid libraries that do not provide explicit interfaces.	
		Carefully construct explicit interfaces for the library procedures where library modules are not provided.	
6.50	Unanticipated exceptions from library routines		
		Translate exceptions into Fortran conformant status values and handle each error situation.	
		Check any return flags present and, if an error is indicated, take appropriate actions when calling a library procedure.	

6.51	Pre-processor directives		
		Avoid use of the C pre-processor `cpp`.	
		Avoid pre-processors generally. Where deemed necessary, a Fortran mode should be set.	
		Use processor-specific modules in place of pre-processing wherever possible.	
6.52	Suppression of language-defined checks		
		Use all run-time checks that are available during development.	
		Use all run-time checks that are available during production running, except where performance is critical.	
		Use several processors during development to check as many conditions as possible.	
6.53	Provision of inherently unsafe operations		
		Provide an explicit interface for each external procedure or replace the procedure by an internal or module procedure.	
		Avoid the use of the intrinsic function `transfer`.	
		Avoid the use of `common` and `equivalence`.	
		Use the compiler or other automatic tool for checking the types of the arguments in calls between Fortran and C, make use of them during development and in production running except where performance would be severely affected.	
6.54	Obscure language features		
		Use the processor or other static analysis tools to detect and identify obsolescent or deleted features and replace them by better methods.	
		Avoid the use of `common` and `equivalence`.	
		Avoid explicit and implicit usages of the `save` attribute in recursive invocations of a procedure and in `do concurrent` constructs.	
		Specify the `save` attribute when supplying an initial value.	
		Use `implicit none` to require explicit declarations.	
6.55	Unspecified behaviour - No Fortran-specific guidance		
6.56	Undefined behaviour		
		Use processor options to detect and report use of non-standard features.	
		Obtain diagnostics from more than one source, for example, use code checking tools or multiple compilers.	

		Supply an explicit interface to specify the `external` attribute for all external procedures invoked.	
		Avoid use of non-standard intrinsic procedures.	
		Specific the `intrinsic` attribute for all non-standard intrinsic procedures and modules referenced.	
6.57	Implementation-defined behaviour		
		Use processor options to detect and report use of non-standard features.	
		Obtain diagnostics from more than one source, for example, use code checking tools or multiple Fortran compilers.	
		Supply an explicit interface to specify the `external` attribute for all external procedures invoked.	
		Avoid use of non-standard intrinsic procedures.	
		Specific the `intrinsic` attribute for all non-standard intrinsic procedures and modules referenced.	
6.58	Deprecated language features		
		Use the processor to detect and identify obsolescent or deleted features and replace them by better methods.	
6.59	Concurreny -- Activation - nothing specific		
6.60	Concurrency -- directed termination - N/A		
6.61	Concurrent data access		
		Use coarrays only when communication among images is necessary.	
		Use one or more of the following mechanisms to ensure correct execution when executing on more than one image,	
		• Use the `sync_all` statement to separate the alteration of the value of a coarray variable on one image from its access by any other image.	
		• Use the `sync_images` statement to separate the alteration of the value of a coarray variable on one image from its access by an image in a specified set of images.	
		• Use a collective subroutine whenever suitable.	

		• Use integer variables of kind `atomic_int_kind` and logical variables of kind `atomic_logical_kind` and use atomic intrinsic subroutines including `atomic_define, atomic_ref,` and `atomic_or` to guarantee sequential access.	
		• Use the `event post` statement in one image and the corresponding `event wait` statement on another image to impose sequential ordering.	
		• Use a critical section to limit execution to one image at a time; if performance using critical sections is unacceptable, use locks and perform analysis to show correct lock behaviour.	
		Avoid	
		• The use of the `volatile` attribute.	
		• The use of the `asynchronous` attribute except for use with a parallel-processing package such as MPI for nonblocking data transfer.	
		• The use of the `sync memory` statement for defining and ordering segments.	
6.62	Concurrency -- premature termination		
		Use the intrinsic functions `failed_images`, `stopped_images`, and `image_status` to detect failed and stopped images.	
		In order to continue execution in the presence of failed images, from time-to-time store relevant information for each team of images externally or on another team, so that the computation can be resumed on a reduced number of images or with images kept in reserve and idle replacing failed images.	
		If continued execution is not desired in the presence of failed images, follow a strategy that ensures safe termination of the executing images.	
6.63	Protocol lock errors		
		Use collective subroutines whenever possible.	
6.64	Uncontrolled format string		
		Wherever possible, use format strings that are constants.	
		Where a variable string is needed, include code to check that its value is within expectations.	
6.65	Modifying constants		

		Always use intent specifications for dummy arguments.	
		Avoid a pointer to an `in` dummy argument.	
		Use the compiler or static analysis tools to detect any use of a constant or `in` dummy argument that is not in accord with the Standard.	

DMJ	38	E	

				*
				*
				*
				*

				*