Section 14: Scope, association, and definition

Entities are identified by lexical tokens within a **scope** that is a program, a scoping unit, a construct, a single statement, or part of a statement. If the scope is a program, the entity is called a **global entity**. If the scope is a scoping unit (2.2), the entity is called a **local entity**. If the scope is a construct, the entity is called a **construct entity**. If the scope is a statement or part of a statement, the entity is called a **statement entity**.

An entity may be identified by

(1) A name (14.1),
(2) A label (14.2),
(3) An external input/output unit number (14.3),
(4) An operator symbol (14.4), or
(5) An assignment symbol (14.5).

By means of association, an entity may be referred to by the same identifier or a different identifier in a different scoping unit, or by a different identifier in the same scoping unit.

14.1 Scope of names

Named entities are global, local, construct, or statement entities.

14.1.1 Global entities

Program units, common blocks, and external procedures are global entities of a program. A name that identifies a global entity shall not be used to identify any other global entity in the same program.

NOTE 14.1

Of the various types of procedures, only external procedures have global names. An implementation may wish to assign global names to other entities in the Fortran program such as internal procedures, intrinsic procedures, procedures implementing intrinsic operators, procedures implementing input/output operations, etc. If this is done, it is the responsibility of the processor to ensure that none of these names conflicts with any of the names of the external procedures, with other globally named entities in a standard-conforming program, or with each other. For example, this might be done by including in each such added name a character that is not allowed in a standard-conforming name or by using such a character to combine a local designation with the global name of the program unit in which it appears.

14.1.2 Local entities

Within a scoping unit, entities in the following classes:

(1) Named variables that are not statement or construct entities (14.1.3), named constants, named constructs, statement functions, internal procedures, module procedures, dummy procedures, intrinsic procedures, generic identifiers, derived types, and namelist group names,
(2) Type components, in a separate class for each type, and
(3) Argument keywords, in a separate class for each procedure with an explicit interface

are local entities of that scoping unit.

Except for a common block name (14.1.2.1), an external procedure name that is also a generic name (12.3.2.1), or an external function name within its defining subprogram (14.1.2.2), a name that identifies a global entity in a scoping unit shall not be used to identify a local entity of class (1) in that scoping unit.

Within a scoping unit, a name that identifies a local entity of one class shall not be used to identify another local entity of the same class, except in the case of generic names (12.3.2.1). A name that identifies a local entity of one class may be used to identify a local entity of another class.

NOTE 14.2

An intrinsic procedure is inaccessible in a scoping unit containing another local entity of the same class and having the same name. For example, in the program fragment

```
SUBROUTINE SUB
   ...
   A = SIN (K)
   ...
CONTAINS
   FUNCTION SIN (X)
      ...
   END FUNCTION SIN
END SUBROUTINE SUB
```

any reference to function SIN in subroutine SUB refers to the internal function SIN, not to the intrinsic function of the same name.

The name of a local entity identifies that entity in a scoping unit and may be used to identify any local or global entity in another scoping unit except in the following cases:

(1) The name that appears as a *subroutine-name* in a *subroutine-stmt* has limited use within the scope established by the *subroutine-stmt*. It can be used to identify recursive references of the subroutine or to identify the name of a common block (the latter is possible only for internal and module subroutines).

(2) The name that appears as a *function-name* in a *function-stmt* has limited use within the scope established by that *function-stmt*. It can be used to identify the result variable, to identify recursive references of the function, or to identify the name of a common block (the latter is possible only for internal and module functions).

(3) The name that appears as an *entry-name* in an *entry-stmt* has limited use within the scope of the subprogram in which the *entry-stmt* appears. It can be used to identify the result variable if the subprogram is a function, to identify recursive references, or to identify the name of a common block (the latter is possible only if the *entry-stmt* is in a module subprogram).

14.1.2.1 Common blocks

A common block name in a scoping unit also may be the name of any local entity other than a named constant, intrinsic procedure, or a local variable that is also an external function in a function subprogram. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity.

NOTE 14.3

An intrinsic procedure name may be a common block name in a scoping unit that does not reference the intrinsic procedure.

14.1.2.2 Function results

For each FUNCTION statement or ENTRY statement in a function subprogram, there is a result variable. If there is no RESULT clause, the result variable has the same name as the function being defined; otherwise, the result variable has the name specified in the RESULT clause.

14.1.2.3 Unambiguous generic procedure references

This subsection contains the rules that shall be satisfied by every pair of specific procedures that have the same generic name, have the same generic operator, or both define assignment. They ensure that a generic reference is unambiguous. When an intrinsic operator or assignment is extended, the rules apply as if the intrinsic consisted of a collection of specific procedures, one for each allowed combination of type, kind type parameter, and rank for each operand. When a generic procedure is accessed from a module, the rules apply to all the specific versions even if some of them are inaccessible by their specific names. If two or more generic interfaces that are accessible in a scoping unit have the same local name, the same operator, or are both assignments, they are interpreted as a single generic interface.

Within a scoping unit, if two procedures have the same generic operator and the same number of arguments or both define assignment, one shall 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.

Within a scoping unit, two procedures that have the same generic name shall both be subroutines or both be functions, and

(1) one of them shall have more nonoptional dummy arguments of a particular data type, kind type parameter, and rank than the other has dummy arguments (including optional dummy arguments) of that data type, kind type parameter, and rank; or

(2) at least one of them shall have both

   (a) A nonoptional dummy argument that corresponds by position in the argument list to a dummy argument not present in the other, present with a different type, present with a different kind type parameter, or present with a different rank; and

   (b) A nonoptional dummy argument that corresponds by argument keyword to a dummy argument not present in the other, present with a different type, present with a different kind type parameter, or present with a different rank.

   Further, the dummy argument that disambiguates by position shall either be the same as or occur earlier in the argument list than the one that disambiguates by keyword.

If a generic name is the same as the name of a generic intrinsic procedure, the generic intrinsic procedure is not accessible if the procedures in the interface and the intrinsic procedure are not all functions or not all subroutines. If a generic invocation applies to both a specific procedure from an interface and an accessible generic intrinsic procedure, it is the specific procedure from the interface that is referenced.

NOTE 14.4

The procedures with interface bodies given by the interface block

```
INTERFACE A
   SUBROUTINE AR (X)
      REAL X
   END SUBROUTINE AR

   SUBROUTINE AI (J)
      INTEGER J
   END SUBROUTINE AI
END INTERFACE A
```

NOTE 14.4 ***(Continued)***

> satisfy rules (2)(a) and (2)(b). However, if J were declared REAL, rule (2)(a) would not be satisfied while rule (2)(b) remains satisfied; in this case, the reference to A in the statement
>
> `CALL A (0.0)`
>
> would be ambiguous.

NOTE 14.5

> If a reference to the intrinsic function NULL appears as an actual argument in a reference to a generic procedure, the argument MOLD may be required to resolve the reference (7.1.4.1).

14.1.2.4 Resolving procedure references

The rules for interpreting a procedure reference depend on whether the procedure name in the reference is established by the available declarations and specifications to be generic in the scoping unit containing the reference, is established to be only specific in the scoping unit containing the reference, or is not established.

(1) A procedure name is established to be generic in a scoping unit

- (a) if that scoping unit contains an interface block with that name;
- (b) if that scoping unit contains an INTRINSIC attribute specification for that name and it is the name of a generic intrinsic procedure;
- (c) if that scoping unit contains a USE statement that makes that procedure name accessible and the corresponding name in the module is established to be generic; or
- (d) if that scoping unit contains no declarations of that name, that scoping unit has a host scoping unit, and that name is established to be generic in the host scoping unit.

(2) A procedure name is established to be only specific in a scoping unit if it is established to be specific and not established to be generic. It is established to be specific

- (a) if that scoping unit contains an interface body with that name;
- (b) if that scoping unit contains a module subprogram, internal subprogram, or statement function that defines a procedure with that name;
- (c) if that scoping unit contains an INTRINSIC attribute specification for that name and if it is the name of a specific intrinsic procedure;
- (d) if that scoping unit contains an EXTERNAL attribute specification for that name;
- (e) if that scoping unit contains a USE statement that makes that procedure name accessible and the corresponding name in the module is established to be specific; or
- (f) if that scoping unit contains no declarations of that name, that scoping unit has a host scoping unit, and that name is established to be specific in the host scoping unit.

(3) A procedure is not established in a scoping unit if it is neither established to be generic nor established to be specific.

14.1.2.4.1 Resolving procedure references to names established to be generic

(1) If the reference is consistent with one of the specific interfaces of a generic interface that has that name and either is in the scoping unit in which the reference appears or is made accessible by a USE statement in the scoping unit, the reference is to the specific procedure in that interface block that provides that interface. The rules in 14.1.2.3 ensure that there can be at most one such specific procedure.

(2) If (1) does not apply, if the reference is consistent with an elemental reference to one of the specific interfaces of a generic interface that has that name and either is in the

scoping unit in which the reference appears or is made accessible by a USE statement in the scoping unit, the reference is to the specific elemental procedure in that interface block that provides that interface. The rules in 14.1.2.3 ensure that there can be at most one such specific interface.

NOTE 14.6

These rules allow specific instances of a generic function to be used for specific array ranks and a general elemental version to be used for other ranks. Given an interface block such as:

```
INTERFACE RANF
          ELEMENTAL FUNCTION SCALAR_RANF(X)
          REAL X
          END FUNCTION SCALAR_RANF

          FUNCTION VECTOR_RANDOM(X)
          REAL X(:)
          REAL VECTOR_RANDOM(SIZE(X))
          END FUNCTION VECTOR_RANDOM
END INTERFACE RANF
```

and a declaration such as:

```
REAL A(10,10), AA(10,10)
```

then the statement

```
A = RANF(AA)
```

is an elemental reference to SCALAR_RANF. The statement

```
A = RANF(AA(6:10,2))
```

is a nonelemental reference to VECTOR_RANDOM.

(3) If (1) and (2) do not apply, if the scoping unit contains either an INTRINSIC attribute specification for that name or a USE statement that makes that name accessible from a module in which the corresponding name is specified to have the INTRINSIC attribute, and if the reference is consistent with the interface of that intrinsic procedure, the reference is to that intrinsic procedure.

NOTE 14.7

In the USE statement case, it is possible, because of the renaming facility, for the name in the reference to be different from the name of the intrinsic procedure.

(4) If (1), (2), and (3) do not apply, if the scoping unit has a host scoping unit, if the name is established to be generic in that host scoping unit, and if there is agreement between the scoping unit and the host scoping unit as to whether the name is a function name or a subroutine name, the name is resolved by applying the rules in this section to the host scoping unit.

14.1.2.4.2 Resolving procedure references to names established to be only specific

(1) If the scoping unit contains an interface body or EXTERNAL attribute specification for the name, if the scoping unit is a subprogram, and if the name is the name of a dummy argument of that subprogram, the dummy argument is a dummy procedure and the reference is to that dummy procedure. That is, the procedure invoked by executing that reference is the procedure supplied as the actual argument corresponding to that dummy procedure.

(2) If the scoping unit contains an interface body or EXTERNAL attribute specification for the name and if (1) does not apply, the reference is to an external procedure with that name.

(3) If the scoping unit contains a module subprogram, internal subprogram, or statement function that defines a procedure with the name, the reference is to the procedure so defined.

(4) If the scoping unit contains an INTRINSIC attribute specification for the name, the reference is to the intrinsic with that name.

(5) If the scoping unit contains a USE statement that makes a procedure accessible by the name, the reference is to that procedure.

NOTE 14.8

Because of the renaming facility of the USE statement, the name in the reference may be different from the original name of the procedure.

(6) If none of the above apply, the scoping unit shall have a host scoping unit, and the reference is resolved by applying the rules in this section to the host scoping unit.

14.1.2.4.3 Resolving procedure references to names not established

(1) If the scoping unit is a subprogram and if the name is the name of a dummy argument of that subprogram, the dummy argument is a dummy procedure and the reference is to that dummy procedure. That is, the procedure invoked by executing that reference is the procedure supplied as the actual argument corresponding to that dummy procedure.

(2) If (1) does not apply, if the name is the name of an intrinsic procedure, and if there is agreement between the reference and the status of the intrinsic procedure as being a function or subroutine, the reference is to that intrinsic procedure.

(3) If (1) and (2) do not apply, the reference is to an external procedure with that name.

14.1.2.5 Components

A component name has the same scope as the type of which it is a component. Outside the type definition, it may appear only within a designator of a component of a structure of that type. If the type is accessible in another scoping unit by use association or host association (14.6.1.2, 14.6.1.3) and the definition of the type does not contain the PRIVATE statement (4.4.1), the component name is accessible for names of components of structures of that type in that scoping unit.

14.1.2.6 Argument keywords

A dummy argument name in an internal procedure, module procedure, or a procedure interface block has a scope as an argument keyword of the scoping unit of the host of the procedure or interface block. As an argument keyword, it may appear only in a procedure reference for the procedure of which it is a dummy argument. If the procedure or procedure interface block is accessible in another scoping unit by use association or host association (14.6.1.2, 14.6.1.3), the argument keyword is accessible for procedure references for that procedure in that scoping unit.

A dummy argument name in an intrinsic procedure has a scope as an argument keyword of the scoping unit making reference to the procedure. As an argument keyword, it may appear only in a procedure reference for the procedure of which it is a dummy argument.

14.1.3 Statement and construct entities

The name of a variable that appears as the DO variable of an implied-DO in a DATA statement or an array constructor has a scope of the implied-DO list. It is a scalar variable that has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the DATA statement or array constructor, and this type shall be integer type; it has no other attributes.

The name of a variable that appears as an index-name in a FORALL statement or FORALL construct has a scope of the statement or construct. It is a scalar variable that has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the FORALL, and this type shall be integer type; it has no other attributes.

The name of a variable that appears as a dummy argument in a statement function statement has a scope of the statement in which it appears. It has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the statement function.

Except for a common block name or a scalar variable name, a name that identifies a global entity or local entity of class 1 (14.1.2) accessible in the scoping unit that contains a statement shall not be the name of a statement entity of that statement. Within the scope of a statement entity, another statement entity shall not have the same name.

If the name of a global or local entity accessible in the scoping unit of a statement is the same as the name of a statement entity in that statement, the name is interpreted within the scope of the statement entity as that of the statement entity. Elsewhere in the scoping unit, including parts of the statement outside the scope of the statement entity, the name is interpreted as that of the global or local entity.

Except for a common block name or a scalar variable name, a name that identifies a global entity or a local entity of class 1 (14.1.2) accessible in the scoping unit of a FORALL statement or FORALL construct shall not be the same as the *index-name*. Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct shall not have the same *index-name*.

If the name of a global or local entity accessible in the scoping unit of a FORALL statement or FORALL construct is the same as the *index-name*, the name is interpreted within the scope of the FORALL statement or FORALL construct as that of the *index-name*. Elsewhere in the scoping unit, the name is interpreted as that of the global or local entity.

14.2 Scope of labels

A label is a local entity. No two statements in the same scoping unit may have the same label.

14.3 Scope of external input/output units

An external input/output unit is a global entity.

14.4 Scope of operators

The intrinsic operators are global entities. A defined operator that is not an extended intrinsic operator is a local entity. Within a scoping unit an operator may identify additional operations as specified by the rules for generic operators (12.3.2.1).

14.5 Scope of the assignment symbol

The assignment symbol is a global entity. Within a scoping unit the assignment symbol may identify additional assignment operations or replace the intrinsic derived type assignment operation as specified by the rules for generic assignment (12.3.2.1).

14.6 Association

Two entities may become associated by name association, pointer association, or storage association.

14.6.1 Name association

There are three forms of **name association** : argument association, use association, and host association. Argument, use, and host association provide mechanisms by which entities known in one scoping unit may be accessed in another scoping unit.

14.6.1.1 Argument association

The rules governing argument association are given in Section 12. As explained in 12.4, execution of a procedure reference establishes an association between an actual argument and its corresponding dummy argument. Argument association may be sequence association (12.4.1.4).

The name of the dummy argument may be different from the name, if any, of its associated actual argument. The dummy argument name is the name by which the associated actual argument is known, and by which it may be accessed, in the referenced procedure.

NOTE 14.9

An actual argument may be a nameless data entity, such as an expression that is not simply a variable or constant.

Upon termination of execution of a procedure reference, all argument associations established by that reference are terminated. A dummy argument of that procedure may be associated with an entirely different actual argument in a subsequent invocation of the procedure.

14.6.1.2 Use association

Use association is the association of names in different scoping units specified by a USE statement. The rules for use association are given in 11.3.2. They allow for the renaming of the entities being accessed. Use association allows access in one scoping unit to entities defined in another scoping unit and remains in effect throughout the execution of the program.

14.6.1.3 Host association

An internal subprogram, a module subprogram, or a derived-type definition has access to the named entities from its host via **host association**. The accessed entities are known by the same name and have the same attributes as in the host and are named data objects, derived types, interface blocks, procedures, generic identifiers (12.3.2.1), and namelist groups.

If an entity that is accessed by use association has the same nongeneric name as a host entity, the host entity is inaccessible by that name. A name that is declared to be an external procedure name by an *external-stmt* or an *interface-body*, or that appears as a *module-name* in a *use-stmt* is a global name and any entity of the host that has this as its nongeneric name is inaccessible by that name. A name that appears in the scoping unit as

(1) A *function-name* in a *stmt-function-stmt* or in an *entity-decl* in a *type-declaration-stmt*;
(2) An *object-name* in an *entity-decl* in a *type-declaration-stmt*, in a *pointer-stmt*, in a *save-stmt*, or in a *target-stmt*;
(3) A *named-constant* in a *named-constant-def* in a *parameter-stmt*;
(4) An *array-name* in an *allocatable-stmt* or in a *dimension-stmt*;
(5) A *variable-name* in a *common-block-object* in a *common-stmt*;
(6) The name of a variable that is wholly or partially initialized in a *data-stmt*;
(7) The name of an object that is wholly or partially equivalenced in an *equivalence-stmt*;
(8) A *dummy-arg-name* in a *function-stmt*, in a *subroutine-stmt*, in an *entry-stmt*, or in a *stmt-function-stmt*;
(9) A *result-name* in a *function-stmt* or in an *entry-stmt*;
(10) An *intrinsic-procedure-name* in an *intrinsic-stmt*;

(11) A *namelist-group-name* in a *namelist-stmt*;

(12) A *generic-name* in a *generic-spec* in an *interface-stmt*; or

(13) The name of a named construct

is the name of a local entity and any entity of the host that has this as its nongeneric name is inaccessible by that name by host association. If a scoping unit contains a subprogram or a derived type definition, the name of the subprogram or derived type is the name of a local entity and any entity of the host that has this as its nongeneric name is inaccessible by that name. Entities that are local (14.1.2) to a subprogram are not accessible to its host.

If a host entity is inaccessible only because a local entity with the same name is wholly or partially initialized in a DATA statement, the local entity shall not be referenced or defined prior to the DATA statement.

If a derived type name of a host is inaccessible, data entities of that type or subobjects of such data entities still can be accessible.

NOTE 14.10

An interface body does not access the named entities by host association, but it may access entities by use association (11.3.2).

If a procedure gains access to a pointer by host association, the association of the pointer with a target that is current at the time the procedure is invoked remains current within the procedure. This pointer association may be changed within the procedure. When execution of the procedure completes, the pointer association that was current remains current, except where the completion causes the target to become undefined (item (3) of 14.7.6). In these cases, the completion of the procedure causes the pointer association status of the host associated pointer to become undefined.

NOTE 14.11

A host subprogram and an internal subprogram may contain the same and differing use-associated entities, as illustrated in the following example.

```
MODULE B; REAL BX, Q; INTEGER IX, JX; END MODULE B
MODULE C; REAL CX; END MODULE C
MODULE D; REAL DX, DY, DZ; END MODULE D
MODULE E; REAL EX, EY, EZ; END MODULE E
MODULE F; REAL FX; END MODULE F
MODULE G; USE F; REAL GX; END MODULE G

PROGRAM A
USE B; USE C; USE D
   ...
CONTAINS
   SUBROUTINE INNER_PROC (Q)
      USE C             ! Not needed
      USE B, ONLY: BX   ! Entities accessible are BX, IX, and JX
                        ! if no other IX or JX
                        ! is accessible to INNER_PROC
                        ! Q is local to INNER_PROC,
                        ! since Q is a dummy argument
      USE D, X => DX    ! Entities accessible are DX, DY, and DZ
                        ! X is local name for DX in INNER_PROC
                        ! X and DX denote same entity if no other
                        ! entity DX is local to INNER_PROC
      USE E, ONLY: EX   ! EX is accessible in INNER_PROC, not    program A
                        ! EY and EZ are not accessible in INNER_PROC
                        ! or in program A
      USE G             ! FX and GX are accessible in INNER_PROC
      ...
```

NOTE 14.11 *(Continued)*

```
   END SUBROUTINE INNER_PROC
END PROGRAM A
```

Because program A contains the statement

```
USE B
```

all of the entities in module B, except for Q, are accessible in INNER_PROC, even though INNER_PROC contains the statement

```
USE B, ONLY: BX
```

The USE statement with the ONLY keyword means that this particular statement brings in only the entity named, not that this is the only variable from the module accessible in this scoping unit.

NOTE 14.12

For more examples of host association, see section C.10.1.

14.6.2 Pointer association

Pointer association between a pointer and a target allows the target to be referenced by a reference to the pointer. At different times during the execution of a program, a pointer may be undefined, associated with different targets, or be disassociated. If a pointer is associated with a target, the definition status of the pointer is either defined or undefined, depending on the definition status of the target.

NOTE 14.13

A pointer from a module program unit may be accessible in a subprogram via use association. Such pointers have a lifetime that is greater than targets that are declared in the subprogram, unless such targets are saved. Therefore, if such a pointer is associated with a local target, there is the possibility that when a procedure defined by the subprogram completes execution, the target will cease to exist, leaving the pointer "dangling". This standard considers such pointers to be in an undefined state. They are neither associated nor disassociated. They shall not be used again in the program until their status has been reestablished. There is no requirement on a processor to be able to detect when a pointer target ceases to exist.

14.6.2.1 Pointer association status

A pointer may have a **pointer association status** of associated, disassociated, or undefined. Its association status may change during execution of a program. Unless a pointer is initialized (explicitly or by default), it has an initial association status of undefined. A pointer may be initialized to have an association status of disassociated.

14.6.2.1.1 Events that cause pointers to become associated

A pointer becomes associated as follows:

(1) The pointer is allocated (6.3.1) as the result of the successful execution of an ALLOCATE statement referencing the pointer, or

(2) The pointer is pointer-assigned to a target (7.5.2) that is associated or is specified with the TARGET attribute and, if allocatable, is currently allocated.

14.6.2.1.2 Events that cause pointers to become disassociated

A pointer becomes disassociated as follows:

(1) The pointer is nullified (6.3.2),

(2) The pointer is deallocated (6.3.3), or

(3) The pointer is pointer-assigned to a disassociated pointer (7.5.2).

(4) The pointer is an ultimate component of an object of a type for which default initialization is specified for the component and
  (a) a function with this object as its result is invoked,
  (b) a procedure with this object as an INTENT (OUT) dummy argument is invoked,
  (c) a procedure with this object as an automatic data object is invoked,
  (d) a procedure with this object as a local object that is not accessed by use or host association is invoked, or
  (e) this object is allocated.

14.6.2.1.3 Events that cause the association status of pointers to become undefined

The following events cause the association status of a pointer to become undefined:

(1) The pointer is pointer-assigned to a target that has an undefined association status,

(2) The target of the pointer is deallocated other than through the pointer,

(3) Execution of a RETURN or END statement that causes the pointer's target to become undefined (item (3) of 14.7.6), or

(4) Execution of a RETURN or END statement in a subprogram where the pointer was either declared or, with the exceptions described in 6.3.3.2, accessed.

14.6.2.2 Pointer definition status

The definition status of a pointer is that of its target. If a pointer is associated with a definable target, the definition status of the pointer may be defined or undefined according to the rules for a variable (14.7).

14.6.2.3 Relationship between association status and definition status

If the association status of a pointer is disassociated or undefined, the pointer shall not be referenced or deallocated. Whatever its association status, a pointer always may be nullified, allocated, or pointer assigned. A nullified pointer is disassociated. When a pointer is allocated, it becomes associated but undefined. When a pointer is pointer assigned, its association and definition status are determined by its target.

14.6.3 Storage association

Storage sequences are used to describe relationships that exist among variables, common blocks, and result variables. **Storage association** is the association of two or more data objects that occurs when two or more storage sequences share or are aligned with one or more storage units.

14.6.3.1 Storage sequence

A **storage sequence** is a sequence of storage units. The **size of a storage sequence** is the number of storage units in the storage sequence. A **storage unit** is a character storage unit, a numeric storage unit, or an unspecified storage unit.

In a storage association context

(1) A nonpointer scalar object of type default integer, default real, or default logical occupies a single **numeric storage unit**;

(2) A nonpointer scalar object of type double precision real or default complex occupies two contiguous numeric storage units;

(3) A nonpointer scalar object of type default character and character length one occupies one **character storage unit**;

(4) A nonpointer scalar object of type default character and character length *len* occupies *len* contiguous character storage units;

(5) A nonpointer scalar object of type nondefault integer, real other than default or double precision, nondefault logical, nondefault complex, nondefault character of any length, or nonsequence type occupies a single **unspecified storage unit** that is different for each case;

(6) A nonpointer array of intrinsic type or sequence derived type occupies a sequence of contiguous storage sequences, one for each array element, in array element order (6.2.2.2);

(7) A nonpointer scalar object of sequence type occupies a sequence of storage sequences corresponding to the sequence of its ultimate components; and

(8) A pointer occupies a single unspecified storage unit that is different from that of any nonpointer object and is different for each combination of type, type parameters, and rank.

A sequence of storage sequences forms a storage sequence. The order of the storage units in such a composite storage sequence is that of the individual storage units in each of the constituent storage sequences taken in succession, ignoring any zero-sized constituent sequences.

Each common block has a storage sequence (5.5.2.1).

14.6.3.2 Association of storage sequences

Two nonzero-sized storage sequences s_1 and s_2 are **storage associated** if the ith storage unit of s_1 is the same as the jth storage unit of s_2. This causes the $(i+k)$th storage unit of s_1 to be the same as the $(j+k)$th storage unit of s_2, for each integer k such that $1 \le i+k \le size\ of\ s_1$ and $1 \le j+k \le size\ of\ s_2$.

Storage association also is defined between two zero-sized storage sequences, and between a zero-sized storage sequence and a storage unit. A zero-sized storage sequence in a sequence of storage sequences is storage associated with its successor, if any. If the successor is another zero-sized storage sequence, the two sequences are storage associated. If the successor is a nonzero-sized storage sequence, the zero-sized sequence is storage associated with the first storage unit of the successor. Two storage units that are each storage associated with the same zero-sized storage sequence are the same storage unit.

NOTE 14.14

Zero-sized objects may occur in a storage association context as the result of changing a parameter. For example, a program might contain the following declarations:

```
INTEGER, PARAMETER :: PROBSIZE = 10
INTEGER, PARAMETER :: ARRAYSIZE = PROBSIZE * 100
REAL, DIMENSION (ARRAYSIZE) :: X
INTEGER, DIMENSION (ARRAYSIZE) :: IX
   ...
COMMON / EXAMPLE / A, B, C, X, Y, Z
EQUIVALENCE (X, IX)
   ...
```

If the first statement is subsequently changed to assign zero to PROBSIZE, the program still will conform to the standard.

14.6.3.3 Association of scalar data objects

Two scalar data objects are storage associated if their storage sequences are storage associated. Two scalar entities are **totally associated** if they have the same storage sequence. Two scalar entities are **partially associated** if they are associated without being totally associated.

The definition status and value of a data object affects the definition status and value of any storage associated entity. An EQUIVALENCE statement, a COMMON statement, or an ENTRY statement may cause storage association of storage sequences.

An EQUIVALENCE statement causes storage association of data objects only within one scoping unit, unless one of the equivalenced entities is also in a common block (5.5.1.1 and 5.5.2.1).

COMMON statements cause data objects in one scoping unit to become storage associated with data objects in another scoping unit.

A named common block is permitted to contain a sequence of differing storage units provided each scoping unit that accesses the common block specifies an identical sequence of storage units. The same rule applies to blank common blocks. If the sizes of the two blank common blocks differ, the sequence of storage units of the shorter block shall be identical to the initial sequence of the storage units of the longer block.

An ENTRY statement in a function subprogram causes storage association of the result variables.

Partial association may exist only between

(1) An object of default character or character sequence type and an object of default character or character sequence type or

(2) An object of default complex, double precision real, or numeric sequence type and an object of default integer, default real, default logical, double precision real, default complex, or numeric sequence type.

For noncharacter entities, partial association may occur only through the use of COMMON, EQUIVALENCE, or ENTRY statements. For character entities, partial association may occur only through argument association or the use of COMMON or EQUIVALENCE statements.

NOTE 14.15

In the example:

```
REAL A (4), B
COMPLEX C (2)
DOUBLE PRECISION D
EQUIVALENCE (C (2), A (2), B), (A, D)
```

the third storage unit of C, the second storage unit of A, the storage unit of B, and the second storage unit of D are specified as the same. The storage sequences may be illustrated as:

```
Storage unit        1     2     3     4     5
                 ----C(1)----|---C(2)----
                       A(1)  A(2)  A(3)  A(4)
                              --B--
                      ------D------
```

A (2) and B are totally associated. The following are partially associated: A (1) and C (1), A (2) and C (2), A (3) and C (2), B and C (2), A (1) and D, A (2) and D, B and D, C (1) and D, and C (2) and D. Although C (1) and C (2) are each storage associated with D, C (1) and C (2) are not storage associated with each other.

Partial association of character entities occurs when some, but not all, of the storage units of the entities are the same.

NOTE 14.16

In the example:

```
CHARACTER A*4, B*4, C*3
EQUIVALENCE (A (2:3), B, C)
```

A, B, and C are partially associated.

A storage unit shall not be explicitly initialized more than once in a program. Explicit initialization overrides default initialization, and default initialization for an object of derived type overrides default initialization for a component of the object (4.4.1). Default initialization may be specified for a storage unit that is storage associated provided the objects or subobjects supplying the default initialization are of the same type and type parameters, and supply the same value for the storage unit.

14.7 Definition and undefinition of variables

A variable may be defined or may be undefined and its definition status may change during execution of a program. An action that causes a variable to become undefined does not imply that the variable was previously defined. An action that causes a variable to become defined does not imply that the variable was previously undefined.

14.7.1 Definition of objects and subobjects

Arrays, including sections, and variables of derived, character, or complex type are objects that consist of zero or more subobjects. Associations may be established between variables and subobjects and between subobjects of different variables. These subobjects may become defined or undefined.

(1) An object is defined if and only if all of its subobjects are defined.

(2) If an object is undefined, at least one (but not necessarily all) of its subobjects are undefined.

14.7.2 Variables that are always defined

Zero-sized arrays and zero-length strings are always defined.

14.7.3 Variables that are initially defined

The following variables are initially defined:

(1) Variables specified to have initial values by DATA statements,

(2) Variables specified to have initial values by type declaration statements,

(3) Nonpointer direct components of variables of a type in which default initialization is specified for those components, provided that the variables are not accessed by use or host association, do not have the ALLOCATABLE attribute or POINTER attribute, and either have the SAVE attribute or are declared in a main program, MODULE, or BLOCK DATA scoping unit, and

(4) Variables that are always defined.

14.7.4 Variables that are initially undefined

All other variables are initially undefined.

14.7.5 Events that cause variables to become defined

Variables become defined as follows:

(1) Execution of an intrinsic assignment statement other than a masked array assignment or FORALL assignment statement causes the variable that precedes the equals to become defined. Execution of a defined assignment statement may cause all or part of the variable that precedes the equals to become defined.

(2) Execution of a masked array assignment or FORALL assignment statement may cause some or all of the array elements in the assignment statement to become defined (7.5.3).

(3) As execution of an input statement proceeds, each variable that is assigned a value from the input file becomes defined at the time that data is transferred to it. (See (4) in 14.7.6.) Execution of a WRITE statement whose unit specifier identifies an internal file causes each record that is written to become defined.

(4) Execution of a DO statement causes the DO variable, if any, to become defined.

(5) Beginning of execution of the action specified by an implied-DO list in an input/output statement causes the implied-DO variable to become defined.

(6) A reference to a procedure causes the entire dummy argument data object to become defined if the entire corresponding actual argument is defined with a value that is not a statement label.

A reference to a procedure causes a subobject of a dummy argument to become defined if the corresponding subobject of the corresponding actual argument is defined.

(7) Execution of an input/output statement containing an IOSTAT= specifier causes the specified integer variable to become defined.

(8) Execution of a READ statement containing a SIZE= specifier causes the specified integer variable to become defined.

(9) Execution of an INQUIRE statement causes any variable that is assigned a value during the execution of the statement to become defined if no error condition exists.

(10) When a character storage unit becomes defined, all associated character storage units become defined.

When a numeric storage unit becomes defined, all associated numeric storage units of the same type become defined. When an entity of double precision real type becomes defined, all totally associated entities of double precision real type become defined.

When an unspecified storage unit becomes defined, all associated unspecified storage units become defined.

(11) When a default complex entity becomes defined, all partially associated default real entities become defined.

(12) When both parts of a default complex entity become defined as a result of partially associated default real or default complex entities becoming defined, the default complex entity becomes defined.

(13) When all components of a numeric sequence structure or character sequence structure become defined as a result of partially associated objects becoming defined, the structure becomes defined.

(14) Execution of an ALLOCATE or DEALLOCATE statement with a STAT= specifier causes the variable specified by the STAT= specifier to become defined.

(15) Allocation of a zero-sized array causes the array to become defined.

(16) Allocation of an object of a derived type, in which default initialization is specified for any nonpointer direct component, causes that component to become defined.

(17) Invocation of a procedure causes any automatic object of zero size in that procedure to become defined.

(18) Execution of a pointer assignment statement that associates a pointer with a target that is defined causes the pointer to become defined.

(19) Invocation of a procedure that contains a nonsaved local object that is not a dummy argument, is not accessed by use or host association, has neither the ALLOCATABLE nor POINTER attribute, and is of a derived type in which default initialization is specified for any direct components, causes those components of the object to become defined.

(20) Invocation of a procedure that has an INTENT (OUT) dummy argument of a derived type that specifies default initialization for a nonpointer direct component, causes that component of the dummy argument to become defined.

(21) Invocation of a nonpointer function of a derived type, in which default initialization is specified for a nonpointer direct component, causes that component of the function result to become defined.

(22) In a FORALL construct, the *index-name* becomes defined when the *index-name* value set is evaluated.

14.7.6 Events that cause variables to become undefined

Variables become undefined as follows:

(1) When a variable of a given type becomes defined, all associated variables of different type become undefined. However, when a variable of type default real is partially associated with a variable of type default complex, the complex variable does not become undefined when the real variable becomes defined and the real variable does not become undefined when the complex variable becomes defined. When a variable of type default complex is partially associated with another variable of type default complex, definition of one does not cause the other to become undefined.

(2) If the evaluation of a function may cause an argument of the function or a variable in a module or in a common block to become defined and if a reference to the function appears in an expression in which the value of the function is not needed to determine the value of the expression, the argument or variable becomes undefined when the expression is evaluated.

(3) The execution of a RETURN statement or an END statement within a subprogram causes all variables local to its scoping unit or local to the current instance of its scoping unit for a recursive invocation to become undefined except for the following:

  (a) Variables with the SAVE attribute.

  (b) Variables in blank common.

  (c) Variables in a named common block that appears in the subprogram and appears in at least one other scoping unit that is making either a direct or indirect reference to the subprogram.

  (d) Variables accessed from the host scoping unit.

  (e) Variables accessed from a module that also is referenced directly or indirectly by at least one other scoping unit that is making either a direct or indirect reference to the subprogram.

  (f) Variables in a named common block that are initially defined (14.7.3) and that have not been subsequently defined or redefined.

(4) When an error condition or end-of-file condition occurs during execution of an input statement, all of the variables specified by the input list or *namelist-group* of the statement become undefined.

(5) When an error condition, end-of-file condition, or end-of-record condition occurs during execution of an input/output statement and the statement contains any implied-DOs, all of the implied-DO variables in the statement become undefined (9.4.3).

(6) Execution of a defined assignment statement may leave all or part of the variable that precedes the equals undefined.

(7) Execution of a direct access input statement that specifies a record that has not been written previously causes all of the variables specified by the input list of the statement to become undefined.

(8) Execution of an INQUIRE statement may cause the NAME=, RECL=, and NEXTREC= variables to become undefined (9.6).

(9) When a character storage unit becomes undefined, all associated character storage units become undefined.

When a numeric storage unit becomes undefined, all associated numeric storage units become undefined unless the undefinition is a result of defining an associated numeric storage unit of different type (see (1) above).

When an entity of double precision real type becomes undefined, all totally associated entities of double precision real type become undefined.

When an unspecified storage unit becomes undefined, all associated unspecified storage units become undefined.

(10) When an allocatable array is deallocated, it becomes undefined.

(11) Successful execution of an ALLOCATE statement for a non-zero-sized object for which default initialization has not been specified causes the object to become undefined.

(12) Execution of an INQUIRE statement causes all inquiry specifier variables to become undefined if an error condition exists, except for the variable in the IOSTAT= specifier, if any.

(13) When a procedure is invoked

   (a) An optional dummy argument that is not associated with an actual argument is undefined;

   (b) A dummy argument with INTENT (OUT) is undefined except for any nonpointer direct components of the argument for which default initialization is specified;

   (c) An actual argument associated with a dummy argument with INTENT (OUT) becomes undefined;

   (d) A subobject of a dummy argument that does not have INTENT (OUT) is undefined if the corresponding subobject of the actual argument is undefined; and

   (e) The result variable of a function is undefined except for those nonpointer direct components of the result for which default initialization is specified.

(14) When the association status of a pointer becomes undefined or disassociated (6.3), the pointer becomes undefined.

(15) When the execution of a FORALL construct has completed, the *index-name* becomes undefined.