Section 11: Program units

The terms and basic concepts of program units were introduced in 2.2. A program unit is a main program, an external subprogram, a module, or a block data program unit.

This section describes all of these program units except external subprograms, which are described in Section 12.

11.1 Main program

A **main program** is a program unit that does not contain a SUBROUTINE, FUNCTION, MODULE, or BLOCK DATA statement as its first statement.

R1101	*main-program*	**is**	[*program-stmt*] [*specification-part*] [*execution-part*] [*internal-subprogram-part*] *end-program-stmt*
R1102	*program-stmt*	**is**	PROGRAM *program-name*
R1103	*end-program-stmt*	**is**	END [PROGRAM [*program-name*]]

C1101 (R1101) In a *main-program*, the *execution-part* shall not contain a RETURN statement or an ENTRY statement.

C1102 (R1101) The *program-name* may be included in the *end-program-stmt* only if the optional *program-stmt* is used and, if included, shall be identical to the *program-name* specified in the *program-stmt*.

C1103 (R1101) An automatic object shall not appear in the *specification-part* (R204) of a main program.

The **program name** is global to the program, and shall not be the same as the name of any other program unit, external procedure, or common block in the program, nor the same as any local name in the main program.

NOTE 11.1

For explanatory information about uses for the program name, see section C.8.1.

NOTE 11.2

An example of a main program is:

```
PROGRAM ANALYSE
   REAL A, B, C (10,10)     !  Specification part
   CALL FIND                !  Execution part
CONTAINS
   SUBROUTINE FIND          !  Internal subprogram
       ...
   END SUBROUTINE FIND
END PROGRAM ANALYSE
```

The main program may be defined by means other than Fortran; in that case, the program shall not contain a *main-program program-unit*.

11.1.1 Main program executable part

The sequence of *execution-part* statements specifies the actions of the main program during program execution. Execution of a program (R201) begins with the first executable construct of the main program.

A main program shall not be recursive; that is, a reference to it shall not appear in any program unit in the program, including itself.

Normal execution of a program ends with execution of the *end-program-stmt* of the main program or with execution of a STOP statement in any program unit of the program. Execution may also be terminated if certain error conditions occur.

11.1.2 Main program internal subprograms

Any internal subprograms in the main program shall follow the CONTAINS statement. Internal subprograms are described in 12.1.2.2. The main program is called the **host** of its internal subprograms.

11.2 External subprograms

External subprograms are described in Section 12.

11.3 Modules

A **module** contains specifications and definitions that are to be accessible to other program units. A module that is provided as an inherent part of the processor is an **intrinsic module**. A **nonintrinsic module** is defined by a module program unit or a means other than Fortran.

Procedures and types defined in an intrinsic module are not themselves intrinsic.

```
R1104  module                 is  module-stmt
                                      [ specification-part ]
                                      [ module-subprogram-part ]
                                      end-module-stmt

R1105  module-stmt            is  MODULE module-name

R1106  end-module-stmt        is  END [ MODULE [ module-name ] ]
```

C1104 (R1104) If the *module-name* is specified in the *end-module-stmt*, it shall be identical to the *module-name* specified in the *module-stmt*.

C1105 (R1104) A module *specification-part* shall not contain a *stmt-function-stmt*, an *entry-stmt*, or a *format-stmt*.

C1106 (R1104) An automatic object shall not appear in the *specification-part* (R204) of a module.

C1107 (R1104) If an object of a type for which *component-initialization* is specified (R435) appears in the *specification-part* of a module and does not have the ALLOCATABLE or POINTER attribute, the object shall have the SAVE attribute.

The module name is global to the program, and shall not be the same as the name of any other program unit, external procedure, or common block in the program, nor be the same as any local name in the module.

NOTE 11.3

Although statement function definitions, ENTRY statements, and FORMAT statements shall not appear in the specification part of a module, they may appear in the specification part of a module subprogram in the module.

A module is host to any module subprograms (12.1.2.2) it contains, and the entities in the module are therefore accessible in the module subprograms through host association.

NOTE 11.4

For a discussion of the impact of modules on dependent compilation, see section C.8.2.

NOTE 11.5

For examples of the use of modules, see section C.8.3.

11.3.1 Module reference

A USE statement specifying a module name is a **module reference**. At the time a module reference is processed, the public portions of the specified module shall be available. A module shall not reference itself, either directly or indirectly.

The accessibility, public or private, of specifications and definitions in a module to a scoping unit making reference to the module may be controlled in both the module and the scoping unit making the reference. In the module, the PRIVATE statement, the PUBLIC statement (5.2.1), their equivalent attributes (5.1.2.1), and the PRIVATE statement in a derived-type definition (4.5.1) are used to control the accessibility of module entities outside the module.

NOTE 11.6

For a discussion of the impact of accessibility on dependent compilation, see section C.8.2.2.

In a scoping unit making reference to a module, the ONLY option in the USE statement may be used to further limit the accessibility, in that referencing scoping unit, of the public entities in the module.

11.3.2 The USE statement and use association

The **USE statement** provides the means by which a scoping unit accesses named data objects, derived types, type aliases, interface blocks, procedures, abstract interfaces, generic identifiers (12.3.2.1), and namelist groups in a module. The entities in the scoping unit are said to be **use associated** with the entities in the module. The accessed entities have the attributes specified in the module. The entities made accessible are identified by the names or generic identifiers used to identify them in the module. By default, they are identified by the same identifiers in the scoping unit containing the USE statement, but it is possible to specify that different local identifiers be used.

R1107	*use-stmt*	**is**	USE [[, *module-nature*] ::] *module-name* [, *rename-list*]
		or	USE [[, *module-nature*] ::] *module-name* , ■ ■ ONLY : [*only-list*]
R1108	*module-nature*	**is**	INTRINSIC
		or	NON_INTRINSIC
R1109	*rename*	**is**	*local-name* => *use-name*
		or	OPERATOR (*local-defined-operator*) => ■ ■ OPERATOR (*use-defined-operator*)
R1110	*only*	**is**	*generic-spec*
		or	*only-use-name*

		or	*rename*
R1111	*only-use-name*	**is**	*use-name*

C1108 (R1107) If *module-nature* is INTRINSIC, *module-name* shall be the name of an intrinsic module.

C1109 (R1107) If *module-nature* is NON_INTRINSIC, *module-name* shall be the name of a nonintrinsic module.

C1110 (R1105) Each *generic-spec* shall be a public entity in the module.

C1111 (R1107) Each *use-name* shall be the name of a public entity in the module.

R1112	*local-defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*
R1113	*use-defined-operator*	**is**	*defined-unary-op*
		or	*defined-binary-op*

C1112 (R1113) Each *use-defined-operator* shall be a public entity in the module.

A *use-stmt* without a *module-nature* provides access either to an intrinsic or to a nonintrinsic module. If the *module-name* is the name of both an intrinsic and a nonintrinsic module, the nonintrinsic module is accessed.

The USE statement without the ONLY option provides access to all public entities in the specified module.

A USE statement with the ONLY option provides access only to those entities that appear as *generic-spec*s, *use-name*s, or *use-defined-operators* in the *only-list*.

More than one USE statement for a given module may appear in a scoping unit. If one of the USE statements is without an ONLY qualifier, all public entities in the module are accessible. If all the USE statements have ONLY qualifiers, only those entities in one or more of the *only-list*s are accessible.

An accessible entity in the referenced module has one or more local identifiers. These identifiers are

(1) The identifier of the entity in the referenced module if that identifier appears as an *only-use-name* or as the *defined-operator* of a *generic-spec* in any *only* for that module,

(2) Each of the *local-name*s or *local-defined-operator*s the entity is given in any *rename* for that module, and

(3) The identifier of the entity in the referenced module if that identifier does not appear as a *use-name* or *use-defined-operator* in any *rename* for that module.

Two or more accessible entities, other than generic interfaces or defined operators, may have the same identifier only if the identifier is not used to refer to an entity in the scoping unit. Generic interfaces and defined operators are handled as described in section 16.1.2.3. Except for these cases, the local identifier of any entity given accessibility by a USE statement shall differ from the local identifiers of all other entities accessible to the scoping unit through USE statements and otherwise.

NOTE 11.7

There is no prohibition against a *use-name* or *use-defined-operator* appearing multiple times in one USE statement or in multiple USE statements involving the same module. As a result, it is possible for one use-associated entity to be accessible by more than one local identifier.

The local identifier of an entity made accessible by a USE statement shall not appear in any other nonexecutable statement that would cause any attribute (5.1.2) of the entity to be specified in the scoping unit that contains the USE statement, except that it may appear in a PUBLIC or PRIVATE statement in the scoping unit of a module and it may be given the ASYNCHRONOUS or VOLATILE attribute.

NOTE 11.8

The constraints in sections 5.5.1, 5.5.2, and 5.4 prohibit the *local-name* from appearing as a *common-block-object* in a COMMON statement, an *equivalence-object* in an EQUIVALENCE statement, or a *namelist-group-name* in a NAMELIST statement, respectively. There is no prohibition against the *local-name* appearing as a *common-block-name* or a *namelist-object*.

The appearance of such a local identifier in a PUBLIC statement in a module causes the entity accessible by the USE statement to be a public entity of that module. If the identifier appears in a PRIVATE statement in a module, the entity is not a public entity of that module. If the local identifier does not appear in either a PUBLIC or PRIVATE statement, it assumes the default accessibility attribute (5.2.1) of that scoping unit.

A procedure with an implicit interface and public accessibility shall explicitly be given the EXTERNAL attribute in the scoping unit of the module; if it is a function, its type and type parameters shall be explicitly declared in a type declaration statement in that scoping unit.

An intrinsic procedure with public accessibility shall explicitly be given the INTRINSIC attribute in the scoping unit of the module or be used as an intrinsic procedure in that scoping unit.

NOTE 11.9

For a discussion of the impact of the ONLY clause and renaming on dependent compilation, see section C.8.2.1.

NOTE 11.10

Examples:

```
USE STATS_LIB
```

provides access to all public entities in the module STATS_LIB.

```
USE MATH_LIB; USE STATS_LIB, SPROD => PROD
```

makes all public entities in both MATH_LIB and STATS_LIB accessible. If MATH_LIB contains an entity called PROD, it is accessible by its own name while the entity PROD of STATS_LIB is accessible by the name SPROD.

```
USE STATS_LIB, ONLY: YPROD; USE STATS_LIB, ONLY : PROD
```

makes public entities YPROD and PROD in STATS_LIB accessible.

```
USE STATS_LIB, ONLY : YPROD; USE STATS_LIB
```

makes all public entities in STATS_LIB accessible.

11.4 Block data program units

A **block data program unit** is used to provide initial values for data objects in named common blocks.

R1114 *block-data* **is** *block-data-stmt*
 [*specification-part*]
 end-block-data-stmt

R1115 *block-data-stmt* **is** BLOCK DATA [*block-data-name*]

R1116 *end-block-data-stmt* **is** END [BLOCK DATA [*block-data-name*]]

C1113 (R1114) The *block-data-name* may be included in the *end-block-data-stmt* only if it was provided in the *block-data-stmt* and, if included, shall be identical to the *block-data-name* in the *block-data-stmt*.

C1114 (R1114) A *block-data specification-part* may contain only USE statements, type declaration statements, IMPLICIT statements, PARAMETER statements, derived-type definitions, and the following specification statements: COMMON, DATA, DIMENSION, EQUIVALENCE, INTRINSIC, POINTER, SAVE, and TARGET.

C1115 (R1114) A type declaration statement in a *block-data specification-part* shall not contain ALLOCATABLE, EXTERNAL, or BIND attribute specifiers.

NOTE 11.11

For explanatory information about the uses for the *block-data-name*, see section C.8.1.

If an object in a named common block is initially defined, all storage units in the common block storage sequence shall be specified even if they are not all initially defined. More than one named common block may have objects initially defined in a single block data program unit.

NOTE 11.12

In the example

```
BLOCK DATA INIT
   REAL A, B, C, D, E, F
   COMMON /BLOCK1/ A, B, C, D
   DATA A /1.2/, C /2.3/
   COMMON /BLOCK2/ E, F
   DATA F /6.5/
END BLOCK DATA INIT
```

common blocks BLOCK1 and BLOCK2 both have objects that are being initialized in a single block data program unit. B, D, and E are not initialized but they need to be specified as part of the common blocks.

Only an object in a named common block may be initially defined in a block data program unit.

NOTE 11.13

Objects associated with an object in a common block are considered to be in that common block.

The same named common block shall not be specified in more than one block data program unit in a program.

There shall not be more than one unnamed block data program unit in a program.

NOTE 11.14

An example of a block data program unit is:

```
BLOCK DATA WORK
   COMMON /WRKCOM/ A, B, C (10, 10)
   REAL :: A, B, C
   DATA A /1.0/, B /2.0/, C /100 * 0.0/
END BLOCK DATA WORK
```