CF90 Fortran Language Reference Manual

Volume 1

SR–3902 2.0

Cray Research, Inc.

The CF90 compiler includes United States software patents 5,247,696, 5,257,372, and 5,361,354.

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

Requests for copies of Cray Research, Inc. publications should be sent to the following address:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120
USA

Order desk +1–612–683–5907
Fax number +1–612–452–0141

Cray Research Software Documentation Map

The information on the following pages highlights the major body of documentation available for Cray Research customers. The manuals are organized into categories by product designation:

Product	Description
Programming environments	Cray Research programming environments provide an integrated development environment by combining compilers with development tools and libraries.
UNICOS operating system	The UNICOS operating system provides basic system control and management functions. It provides services, including basic batch and interactive system access for users, operating system interfaces for programmers, networking facilities, and sophisticated resource management for administrators. UNICOS MAX, in conjunction with UNICOS on the host system, provides operating system functionality on CRAY T3D systems.
Networking products	Cray Research networking products provide media connections, network programming access, and batch and interactive access for users across a heterogeneous network of computers.
Operating systems supplementary products	The operating systems supplementary products include a variety of products that are used in conjunction with Cray Research operating systems but are not included on the same media.

To use the map, find the product designation closest to your specific needs or role as a Cray Research system user. Documents listed under other categories also may be of interest to you; documents are listed only once, underneath the product to which they most directly apply. The date on the bottom of the page tells you when the information was last revised.

For more information

In addition to the map, you can use the following publications to find documentation specific to your needs:

- *Software Documentation Ready Reference,* publication SQ–2122, serves as a general index to the Cray Research documentation set. The booklet lists documents and man pages according to topic.
- *Software Overview for Users*, publication SG–2052, provides an overview of Cray Research software, including operating systems, programming environments, libraries, tools, applications, and so on. It directs you to documentation that contains user-level information.
- *User Publications Catalog*, publication CP–0099, briefly describes all Cray Research manuals available to you, including some not shown on the map. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on CRInform.

Ordering

To obtain Cray Research publications, order them by publication number from the Distribution Center:

Cray Research, Inc.		
Distribution Center	Order desk	+1–612–683–5907
2360 Pilot Knob Road	Fax number	+1–612–452–0141
Mendota Heights, MN 55120	Email	`orderdsk@cray.com`
USA		

Programming Environments

Ada

Cray Ada Environment, Volume 1: Reference Manual (SR–3014)

Cray Ada Environment, Volume 2: Programming Guide (SR–3082)

Assemblers

Cray Assembler for MPP (CAM) Reference Manual (SR–2510)†

Cray Assembly Language (CAL) for Cray PVP Systems Ready Reference (SQ–3110)

Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual (SR–3108)†

Fortran Compilers

CF90 Ready Reference (SQ–3900)

CF90 Commands and Directives Reference Manual (SR–3901)†

CF90 Fortran Language Reference Manual, Vol. 1 (SR–3902)†

CF90 Fortran Language Reference Manual, Vol. 2 (SR–3903)†

CF90 Fortran Language Reference Manual, Vol. 3 (SR–3905)†

CF90 SPARC Ready Reference (SQ–3904)

Introducing the CF90 SPARC Programming Environment (IN–2155)†

Cray MPP Fortran Reference Manual (SR–2504)†

CF77 Fortran Language Reference Manual (SR–3772)

Cray Standard C Compiler

Cray Standard C Reference Manual (SR–2074)†

Cray Standard C Ready Reference (SQ–2076)

Cray Standard C Reference Manual for MPP (SR–2506)†

C++ Compiler

Cray C/C++ Reference Manual (SR–2179)†

Cray C/C++ Ready Reference (SQ–2180)

CrayLibs

Application Programmer's I/O Guide (SG–2168)†

Application Programmer's Library Reference Manual (SR–2165)*

Introducing CrayLibs (IN–2167)†

Math Library Reference Manual (SR–2138)*

Parallel Virtual Machine (PVM) Reference Card (SQ–2512)

PVM and HeNCE Programmer's Manual (SR–2501)†

Scientific Libraries Reference Manual (SR–2081)*

CrayTools

Compiler Information File (CIF) Reference Manual (SR–2401)†

Cray TotalView Debugger Reference Card (SQ–2519)

Guide to Parallel Vector Applications (SG–2182)†

Introducing the Cray TotalView Debugger (IN–2502)†

Introducing the MPP Apprentice Tool (IN–2511)†

Introducing the Program Browser (IN–2140)†

UNICOS CDBX Symbolic Debugger Reference Card (SQ–2110)

UNICOS CDBX Symbolic Debugger Reference Manual (SR–2091)†

UNICOS CDBX Debugger User's Guide (SG–2094)†

General

Cray Research MPP Software Guide (SG–2508)†

CRAY T3D Emulator User's Guide (SG–2500)†

Installing Programming Environment Products (SG–5191)

Migrating to the CRAY T90 Series IEEE Floating Point (SN–2194)†

Optimizing Code on Cray PVP Systems (SG–2192)†

Programming Environment Installation Guide (SG–5217)

Message Passing Toolkit

MPT: PVM Programmer's Manual (SR–2196)†

MPT: MPI Programmer's Manual (SR–2197)†

† Available with CrayDoc
* Man pages available with the `man` command

UNICOS Operating System

General UNICOS

Software Documentation Ready Reference (SQ–2122)

Software Overview for Users (SG–2052)†

User's Guide to Online Information (SG–2143)†

UNICOS `ed` Reference Card (SQ–2055)

UNICOS Environment Variables Ready Reference (SQ–2117)

UNICOS File Formats and Special Files Reference Manual (SR–2014)★

UNICOS Standard Shell Reference Card (SQ–2115)

UNICOS User Commands Ready Reference (SQ–2056)★

UNICOS User Commands Reference Manual (SR–2011)★

UNICOS `vi` Reference Card (SQ–2054)

UNICOS Visual Interfaces User's Guide (SG–3094)†

Networking Feature Use and Administration

Front-end Protocol Internal Reference Manual (SM–0042)

FTA User and Administrator Manual (SG–2144)†

Kerberos Administrator's Guide (SG–2306)†

Kerberos User's Guide (SG–2409)†

Network Queuing System (NQS) User's Guide (SG–2105)†

ONC+ Technology for the UNICOS Operating System (SG–2169)†

Remote Procedure Call (RPC) Reference Manual (SR–2089)†

TCP/IP Network User's Guide (SG–2009)†

UNICOS Networking Facilities Administrator's Guide (SG–2304)†

UNICOS NQS and NQE Administrator's Guide (SG–2305)†

UNICOS Station Call Processor (USCP) Administrator's Guide (SG–2308)†

UNICOS MAX

Cray MPP Loader User's Guide (SG–2514)†

Cray MPP Simulator User's Guide (SG–2503)†

CRAY T3D Administrator's Guide (SG–2507)†★

UNICOS Basic Administration and Operation

CRAY EL Series IOS Commands Ready Reference (SQ–2162)★

CRAY EL Series IOS Commands Reference Manual (SR–2408)★

CRAY EL Series IOS Messages (SQ–2402)

CRAY IOS-V Commands Reference Manual (SR–2170)★

CRAY IOS-V Messages (SQ–2172)†

General UNICOS System Administration (SG–2301)†

`nmake` Reference Card (SQ–2146)

OLNET Online Diagnostic Network Communications Program Maintenance Manual for UNICOS (SD–1021)†

Online Maintenance Tools Guide for Cray PVP Systems (SD–1012)†

UNICOS Basic Administration Guide for CRAY J90 Series and CRAY EL Series (SG–2416)†

UNICOS Configuration Administrator's Guide (SG–2303)†

UNICOS Installation Guide (SG–2112)

UNICOS Installation Guide for the CRAY EL Series (SG–5201)

UNICOS Installation Guide for the CRAY J90 Series (SG–5271)

UNICOS Installation Menu System Reference Card (SQ–2411)

UNICOS Installation/Configuration Menu System User's Guide (SG–2412)†

UNICOS Resource Administration (SG–2302)†

UNICOS Shared File System (SFS) Administrator's Guide (SG–2114)†

UNICOS Feature Use and Administration

UNICOS Administrator Commands Reference Manual (SR–2022)★

UNICOS Administrator Commands Ready Reference (SQ–2413)★

UNICOS Message System Programmer's Guide (SG–2121)†

UNICOS Multilevel Security (MLS) Feature User's Guide (SG–2111)†

UNICOS System Security Overview for Administrators (SG–2141)†

UNICOS Tape Subsystem Administrator's Guide (SG–2307)†

UNICOS Tape Subsystem User's Guide (SG–2051)†

UNICOS under UNICOS Administrator's Guide (SG–2156)†

† Available with CrayDoc
★ Man pages available with the `man` command

UNICOS Operating System (continued)

Programming Support

Segment Loader (SEGLDR) and ld Reference Manual (SR–0066)†

SEGLDR Reference Card (SQ–0303)

UNICOS System Libraries Ready Reference (SQ–2147)*

UNICOS System Libraries Reference Manual (SR–2080)*

Programming Support (continued)

UNICOS Macros and Opdefs Reference Manual (SR–2403)*

UNICOS System Calls Reference Manual (SR–2012)*

UNICOS X Window System Reference Manual (SR–2101)†

X Window System Resources Ready Reference (SQ–2123)

Networking Products

ATM

Asynchronous Transfer Mode (ATM) Administrator's Guide (SG–2193)†

fy Driver

fy Driver Administrator's Guide (SG–2132)

HIPPI

HIPPI Channel Administrator's Guide (SG–2159)

Network Queuing Environment (NQE)

NQE User's Guide (SG–2148)†

NQE Ready Reference (SG–2149)

NQE Administration (SG–2150)†

Introducing NQE (IN–2153)†

RQS MVS

RQS User's Guide for IBM MVS Systems (SG–2405)

RQS Administrator's Guide for IBM MVS Systems (SG–2406)

Fiber Distributed Data Interface (FDDI)

Fiber Distributed Data Interface (FDDI) Administrator's Guide (SG–3091)

RQS VMS

RQS/NQC User's Guide for DEC OpenVMS/VAX and AXP Systems (SV–3151)

RQS/NQC Administrator's Guide for DEC OpenVMS/VAX and AXP Systems (SV–3152)

SUPERLINK VM

SUPERLINK VM User's Guide: RQS Component (SI–0170)

SUPERLINK VM Administrator's Guide: RQS Component (SI–0171)

SUPERLINK VMS

SUPERLINK Administrator's Guide for DEC VAX/VMS Systems (SV–3154)

SUPERLINK User's Guide for DEC VAX/VMS Systems (SV–3153)

Operating Systems Supplementary Products

CCMT

Concurrent Maintenance Tools (CCMT) Guide (SD–1027)†

Cray/REELlibrarian (CRL)

Cray/REELlibrarian (CRL) Administrator's Guide (SG–2127)

Cray/REELlibrarian (CRL) User's Guide (SG–2126)

SMARTE

System Maintenance and Remote Testing Environment (SMARTE) Guide (SD–1017)†

Data Migration Facility (DMF)

Cray Data Migration Facility (DMF) Administrator's Guide (SG–2135)†*

Cray Data Migration (DMF) MSP Writer's Guide (SN–2098)†

OWS

Support System Reference Manual (SR–3077)*

Support System Operator's Guide (SG–3078)†

Support System Administrator's Guide (SG–3079)†

Support System Ready Reference (SQ–3080)

† Available with CrayDoc
* Man pages available with the man command

New Features

CF90 Fortran Language Reference Manual, Volume 1 SR–3902 2.0

The CF90 compiler now supports CRAY T3E, CRAY J90, and CRAY T90 systems. These new platforms are in addition to the CRAY Y-MP M90, CRAY C90, CRAY EL, CRAY Y-MP E, and SPARC systems that were supported in previous releases. (Note that implementation on CRAY T3E systems is deferred.)

The CF90 2.0 release offers increased performance and many user-interface enhancements.

A statement can be continued with up to 99 continuation lines.

Line lengths in fixed source form have been extended. If `-n 80` is specified on the `f90`(1) command line, statements can appear in columns 7 through 80. If `-n 132` is specified on the `f90`(1) command line, statements can appear in columns 7 through 132.

On SPARC systems, support is now provided for 128-bit floating-point, 64-bit integer, 64-bit logical, and double complex data types.

Boolean (typeless) values larger than one word in length can be used in numeric constants.

Additional forms of octal and hexadecimal literal constants are now supported.

Record of Revision

The date of printing or software version number is indicated in the footer.

Version	***Description***
1.0	December 1993. Original printing.
1.1	June 1994. Online-only revision. Includes minor updates and corrections to revision 1.0.
2.0	October 1995. This printing supports the CF90 compiler release 2.0 running on Cray PVP systems, CRAY T3E systems, and SPARC systems. The implementation of features on CRAY T3E systems is deferred. Sections 11 through 14 of revision 1.0 are now part of the *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903. Appendix sections A through G of revision 1.0 are now part of the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.

Preface

The CF90 compiler implements the Fortran 90 standard. This reference manual describes the Fortran programming language as implemented by the Cray Research CF90 compiler.

The CF90 release 2.0 compiler runs on the following systems:

- CRAY T90, CRAY J90, CRAY C90, CRAY EL, CRAY Y-MP E, CRAY Y-MP M90 systems running the UNICOS 8.0, or later, operating system. This manual refers to these systems as *Cray Research parallel vector systems* or *Cray PVP systems*.
- CRAY T3E systems. This manual sometimes refers to the CRAY T3E system and other Cray Research massively parallel processor systems as *Cray MPP systems*.
- SPARC systems running the Solaris 2.3, or later, operating system. This manual refers to these systems as *SPARC systems*.

> **Note:** This manual describes how the CF90 compiler works on Cray PVP systems, CRAY T3E systems, and SPARC systems. Implementation of the CF90 2.0 compiler on CRAY T3E systems is deferred.

The CF90 compiler was developed to support the Fortran standards adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). These standards, commonly referred to as *the Fortran 90 standard*, are ANSI X3.198–1992 and ISO/IEC 1539:1991. Because the ANSI Fortran 90 standard is a superset of the FORTRAN 77 standard, the CF90 compiler will compile code written to the FORTRAN 77 standard.

> **Note:** The Fortran 90 standard is a substantial revision to the FORTRAN 77 language standard. Because of the number and complexity of the features, the standards organizations are continuing to interpret the Fortran 90 standard for Cray Research and for other vendors. To maintain conformance to the Fortran 90 standard, Cray Research may need to change the behavior of certain CF90 features in future releases based upon the outcomes of the outstanding interpretations to the standard.

Related CF90 publications

This manual is one of a set of Cray Research, Inc. (CRI) manuals that describes the CF90 compiler. The other manuals in this set are as follows:

- *Introducing the CF90 Programming Environment,* publication IN–2155
- *CF90 Ready Reference*, publication SQ–3900
- *CF90 Commands and Directives Reference Manual*, publication SR–3901
- *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903
- *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905
- *CF90 Ready Reference for SPARC Systems*, publication SQ–3904

Within this manual, sections 1 through 10 correspond to sections 1 through 10 of the Fortran 90 standard. Sections 1 through 4 of the *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903, correspond to sections 11 through 14 of the Fortran 90 standard. The *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905, contains CF90 information that supplements the Fortran 90 standard. The standard is the complete official description of the language. This manual set and the standard can be examined in parallel for insights into the Fortran language.

The *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905, contains the complete Fortran 90 syntax in Backus-Naur Form (BNF). The syntax rules are numbered exactly as they are in the Fortran standard. There is a cross reference that lists, for each nonterminal syntactic term, the number of the rule in which it is defined and all rules in which it is referenced.

CF90 messages

You can obtain CF90 compiler message explanations by using the online `explain`(1) command.

CF90 man pages

In addition to printed documentation, several online man pages describe aspects of the CF90 compiler. Man pages exist for the library routines, the intrinsic procedures, and several programming environment tools.

You can print copies of online man pages by using the pipe symbol with the `man`, `col`, and `lpr` commands. In the following example, these commands are used to print a copy of the `f90`(1) man page:

```
man f90 | col -b | lpr
```

Each man page includes a general description of one or more commands, routines, system calls, or other topics, and provides details of their usage (command syntax, routine parameters, system call arguments, and so on). If more than one topic appears on a page, the entry in the printed manual is alphabetized under its primary name; online, secondary entry names are linked to these primary names. For example, `egrep` is a secondary entry on the page with a primary entry name of `grep`. To access `grep` online, you can type `man grep`. To access information about `egrep` online, you can type either `man grep` or `man egrep`; both commands display the `grep` man page on your terminal.

The `-p` and `-s` options to the `dmput`(1) command require that the caller be super user.

Related Fortran publications

The following commercially available reference books are among those that you could consult for more information on the history of Fortran and the Fortran 90 language itself:

- Adams, J., W. Brainerd, J. Martin, B. Smith, and J. Wagener . *Fortran 90 Handbook – Complete ANSI/ISO Reference*. New York, NY: Intertext Publications/Multiscience Press, Inc., 1990.
- Metcalf, M. and J. Reid. *Fortran 90 Explained*. Oxford, UK: Oxford University Press, 1990.
- American National Standards Institute. *American National Standard Programming Language Fortran, ANSI X3.198–1992*. New York, 1992.

- International Standards Organization. ISO/IEC 1539 : 1991, *Information technology – Programming languages – Fortran*. Geneva, 1991.

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software manuals that are available to customers.

Related programming environment publications

The CF90 compiler is part of the Fortran programming environment. The documentation available from Cray Research on the programming environment tools depends on your platform.

- If you are using the CF90 compiler on a SPARC system, you may want to read *Introducing the CF90 Programming Environment*, publication IN–2155. It describes the products in the environment and the Cray Research documentation that is available for them.

- If you are using the CF90 compiler on a Cray PVP system or a CRAY T3E system, the following Cray Research documents describe the products in the environment that may be of interest to you. These publications describe the operating system, input/output (I/O), and other topics:

Publication	Title
SR–0066	*Segment Loader (SEGLDR) and* `ld` *Reference Manual*
SR–201 1	*UNICOS User Commands Reference Manual*
SR–2040	*UNICOS Performance Utilities Reference Manual*
SG–2052	*Software Overview for Users*
SQ–2056	*UNICOS User Commands Ready Reference*
SR–2165	*Application Programmer's Library Reference Manual*
SR–2081	*Scientific Libraries Reference Manual*
SR–2091	*UNICOS CDBX Symbolic Debugger Reference Manual*
SG–2094	*UNICOS CDBX Debugger User's Guide*

Publication	Title
SR–2138	*Math Library Reference Manual*
IN–2140	*Introducing the Program Browser*
SG–2182	*Tuning Guide to Parallel Vector Applications*
IN–2502	*Introducing the TotalView Debugger*
IN–2511	*Introducing the MPP Apprentice Tool*
SN–3009	*Interlanguage Programming Conventions*
SG–2168	*Application Programmer's I/O Guide*
SG–2192	*Optimizing Code on Cray PVP Systems*
SR–2401	*Compiler Information File (CIF) Reference Manual*
SG–3094	*UNICOS Visual Interfaces User's Guide*

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software manuals that are available to customers.

Ordering publications

To order a manual, either call the Distribution Center in Mendota Heights, Minnesota, at +1–612–683–5907 or send a facsimile of your request to fax number +1–612–452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users) or `order desk` (HPDesk users).

If you are a CraySoft customer, you can order additional copies of this product, increase the number of licensed users, or order CraySoft publications, by contacting the CraySoft organization in one of the following ways:

- Call one of these telephone numbers:

  1–800–289–2729

  +1–612–683–3030

- Send mail to the following address:

CraySoft
Cray Research, Inc.
1440 Northland Drive
Mendota Heights, Minnesota 55120
USA

- Send electronic mail from any system connected to Internet to the following address:

`craysoft@cray.com`

- Send a facsimile (fax) to the following telephone number:

+1–612–683–5508

Conventions

This manual describes Fortran statement syntax using a variant of Backus-Naur Form (BNF). The following conventions are used throughout this manual when describing the Fortran statements and other parts of the Fortran language:

Convention	Meaning
`command`	This font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures. `Typewriter` font is also used to represent terminal symbols in parts of BNF.
variable	This typeface denotes variable entries and words or concepts being defined.
`user input`	This font denotes literal items that the user enters in screen drawings of interactive sessions. Output is shown in nonbold `typewriter` font.

Convention	Meaning
[ ]	Brackets enclose optional portions of a command line or syntax statement. The items inside the brackets may be repeated zero or more times.
_list ...	Both *_list* and an ellipses indicate that the preceding item can be repeated. Use of *_list* implies that the members of the list are separated from each other by commas.
*scalar_*	When *scalar_* is the first item in a syntax description, it indicates that the item is a scalar , not an array, value.
_name	When *_name* is part of a syntax definition, it indicates that the item is a name with no qualification. For example, the item must not have a subscript list, so `ARRAY` is a name, but `ARRAY(I)` is not.
(R*nnnn*)	Indicates that the Fortran 90 standard has rules regarding the characteristic of the language being discussed. All rules are numbered, and the numbered list appears in the *CF90 Fortran Reference Manual, Volume 3*, publication SR–3905. The numbering of the rules in this manual matches the numbering of the rules in the standard. The forms of the rules in this manual and the BNF syntax class terms that are used may differ from the rules and terms used in the standard.
`POINTER`	The term `POINTER` refers to the Fortran 90 `POINTER` attribute.
Cray pointer	The term *Cray pointer* refers to the Cray pointer data type extension.

In this publication, *Cray Research*, *Cray*, and *CRI* refer to Cray Research, Inc. and/or its products.

BNF conventions

This subsection describes some of the commonly used BNF conventions:

- Terms such as *goto_stmt* are called *variable entries, nonterminal symbols*, or simply *nonterminals.* The metalanguage term *goto_stmt* , for example, represents the `GO TO` statement, as follows:

  goto_stmt **is** `GO TO` *label*

  The syntax rule defines *goto_stmt* to be `GO TO` *label* , which describes the format of the `GO TO` statement. The description of the `GO TO` statement is not complete until the definition of *label* is given; *label* is also a nonterminal symbol. A further search for *label* will result in a specification of *label* and thereby provide the complete statement definition. A *terminal* part of a syntax rule does not need further definition. For example, `GO TO` is a terminal and is a required part of the statement form. The complete BNF list appears in the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.

- Several abbreviations are commonly used in naming nonterminals. These are as follows:

Abbreviation	Term
arg	argument
attr	attribute
char	character
decl	declaration
def	definition
desc	descriptor
expr	expression
int	integer

Abbreviation	Term
op	operator
spec	specifier or specification
stmt	statement

- The term **is** separates the syntax class name from its definition. The symbol **or** indicates an alternative definition for the syntactic class being defined. The following example shows that *add_op*, the add operator, may be either plus or minus.

add_op	**is**	+
	or	−

- The ≡ symbol indicates a continuation line. If a rule does not fit on one line, the convention is to use the symbol ≡ at the end of the line being continued as well as at the beginning of the line that continues or completes the statement. This is shown in the following example:

allocatable_stmt **is** `ALLOCATABLE [ :: ]` *array_name* `[ (` *deferred_shape_spec_list* `) ]` ≡
≡ `[ ,` *array_name* `[ (` *deferred_shape_spec_list* `) ] ] ...`

Online information

The following types of online information products are available to Cray Research customers:

- CrayDoc online documentation reader, which lets you see the text and graphics of a manual online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Man pages, which describe a particular element of the operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.
- The message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.

- Cray Research online glossary, which explains the terms used in a manual. To get a definition, use the define(1) command. This command is supported on Cray PVP systems and CRAY T3E systems.
- xhelp help facility. This online help system is available within tools such as the Program Browser (xbrowse) and the Cray MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

If you have comments about the technical accuracy, content, or organization of this manual, please tell us. You can contact us in any of the following ways:

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  uunet!cray!publications

- Send us electronic mail from any system connected to Internet, using the following Internet addresses:

  pubs3902@timbuk.cray.com (comments on this manual)

  publications@timbuk.cray.com (general comments)

- Contact your Cray Research representative and ask that a Software Problem Report (SPR) be filed. Use PUBLICATIONS for the group name, PUBS for the command, and NO-LICENSE for the release name.
- CraySoft customers in the United States and Canada can call the CraySoft Customer Support Center at either of the following telephone numbers. These telephone numbers are also listed on the cs_support(7) man page and on the Service and Support card included in the product package.

  1–800–272–9349 (toll free from the United States and Canada)

  +1–770–631–2244

- CraySoft customers outside the United States and Canada can call their local Cray Research support office. These telephone numbers are listed on the cs_support(7) man page and on the Service and Support card contained in the product package.
- Call our Software Publications Group in Eagan, Minnesota, through the Technical Support Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600
- Send a facsimile of your comments to the attention of "Software Publications Group" in Eagan, Minnesota, at fax number +1–612– 683–5599.
- Use the postage-paid Reader's Comment Form at the back of the printed manual.

We value your comments and will respond to them promptly.

Contents

Figures

Tables

Introduction [1]

For a programming language, Fortran has existed for a long time. It was one of the first widely used high-level languages, as well as the first programming language to be standardized. It is still the premier language for scientific and engineering computing applications. This section provides a history of the Fortran language and introduces the Fortran 90 standard.

History

1.1

The following subsections describe the development and standardization of the Fortran language.

Initial development of Fortran

1.1.1

In 1954, a project was begun at International Business Machines Corporation (IBM) under the leadership of John Backus to develop an automatic programming system that would convert programs written in a mathematical notation to machine instructions for the IBM 704 system. Many people were skeptical that the project would be successful, because, at the time, it was felt that computer memories were so small and expensive, and execution time so valuable, that it was necessary for the program produced by the compiler to be almost as efficient as that produced by a good assembly language programmer.

This project produced the first Fortran compiler, which was delivered to a customer in 1957. It was a great success by any reasonable criterion. The efficiency of the code generated by the compiler surprised even some of its authors. A more important achievement, but one that took longer to realize, was that programmers could express their computations in a much more natural way. This increased productivity and permitted programmers to write programs that could be maintained and enhanced much more easily than an assembly language program.

About one year after the introduction of the first Fortran compiler, IBM introduced Fortran II. One of the most important changes in Fortran II was the addition of subroutines that could be compiled independently. Thus, Fortran changed substantially even during its first year; it has been changing continually ever since.

Standardization

1.1.2

By the early 1960s, many computer vendors had implemented a Fortran compiler . They all included special features that were not found in the original IBM compiler. These features usually were included to meet needs and requests of the users and thus provide an inducement for the customer to buy computer systems from the vendor providing the best compiler. Unfortunately , the profusion of dialects of Fortran prevented programs written for one system from being transported to a different computer system.

At about this time, the American Standards Association (ASA), later to become the American National Standards Institute (ANSI), began a project of standardizing many aspects of data processing. Someone had the daring idea of standardizing programming languages. A committee was formed to develop a standard for Fortran under the auspices of the Business Equipment Manufacturers Association (BEMA), later to become the Computer and Business Equipment Manufacturers Association (CBEMA). This standard was adopted in 1966; after the adoption of FORTRAN 77, it became known as FORTRAN 66 to distinguish the two versions.

The language continued to develop after 1966, along with general knowledge in the areas of programming, language design, and computer design. Work on a revision of FORTRAN 66 was completed in 1977 (hence the name FORTRAN 77) and officially published in 1978. The most significant features introduced in this version were the character data type, the `IF-THEN-ELSE` construct, and many new input/output (I/O) facilities such as direct-access files and the `OPEN` statement. Except for the character data type, most of these features had been implemented in many compilers or preprocessors. During this revision, Hollerith data was removed because the character data type is a superior facility.

As soon as the technical development of FORTRAN 77 was completed, ANSI X3J3 (the ANSI Fortran standardization technical subcommittee) and International Standards Organization (ISO) WG5 (the international Fortran standardization technical subcommittee) turned their attention to the next revision, Fortran 90.

The work on Fortran 90 began immediately after the adoption of FORTRAN 77 because, contrary to the pronouncements of some that Fortran was dead, the huge volume of public comments on the proposed standard indicated that there was a tremendous interest in the further development of the language. Many public comments on FORTRAN 77 contained suggestions that have been included in Fortran 90.

Fortran is still the most widely used programming language for scientific and engineering applications, and Fortran 90 continues this tradition.

Fortran, like other programming languages, needs to change over a period of years. Computing technology and programming methodology are evolving at a very rapid pace. Thus, the most obvious reason that programming languages must evolve is that, to be effective, a programmer must have a language that incorporates this new methodology. A good example is provided by control structures. During the 1970s, effort was put into determining the best possible set of control structures that a language should have; this was done mainly from the point-of-view of providing facilities that encourage good program design and ease of program maintenance. In this area, Fortran's lack of modern design was actually a benefit, because Fortran 90 includes a good set of control structures without severely affecting the older control mechanisms already in FORTRAN 77.

In the area of portability, both an awareness of the problem and the technology to cope with it have advanced. As the cost of software maintenance increases and a wider variety of computing systems become available, it is increasingly important to be able to port Fortran programs from one system to another. The main purpose of a standard is to permit portability of programs; in spite of this, several features of each standard Fortran have been nonportable. Each time the standard is revised, features are added to enhance portability and replace features that do not port easily . The most obvious example of this in FORTRAN 77 concerns numeric precision. The precision of real and double-precision values varies greatly from one computer system to the next; when moving from a machine with many digits of precision for real values to a machine with a smaller number of digits, it is often necessary to change many declarations from `REAL` to

DOUBLE PRECISION. This problem was partially addressed in FORTRAN 77 by adding generic intrinsic functions so that function references in the program could remain unchanged; in Fortran 90, numeric quantities can be given a kind parameter that allows a programmer to specify numeric precision requirements in a portable way.

Implementation techniques have improved over time. Language features that required special implementation techniques, such as stack or heap storage management, were avoided because of the implementation cost and the possibility of reducing execution efficiency. Experience with these features in other languages over a long period of time has removed them from the category of features that are difficult to implement efficiently.

Advances in computer architecture, increases in speed, and decreased hardware costs have mitigated some concerns about efficiency. With decreases in computing costs have come increases in personnel costs. The economics of these trends indicate that there should be more features in a language that increase programmer productivity, even if they involve some decrease in machine efficiency.

Changes in hardware architecture have opened up entirely new techniques for solving problems. Probably the most important recent development of this sort in the world of scientific and engineering computing is the use of synchronous parallel processing or vector processing. Some of the fastest machines now available have this sort of architecture. For many algorithms to execute efficiently on these machines, the computations that can be vectorized or performed in parallel must be recognized either by the programmer or by software. The ability of the software to detect parallelism in old Fortran programs is increasing, but there are still many cases in which it is necessary for programmers to indicate these situations. Also, many algorithms involve parallel computations and these are expressed naturally in a language like Fortran 90, which has special provisions, such as the new array processing facilities.

FORTRAN 77 compatibility

1.2

Because of the large investment in existing software written in Fortran, the Fortran standards committee decided to include the entire previous standard (FORTRAN 77) in Fortran 90. Even though the standard describes a category called *deleted* features, there are none; no FORTRAN 77 features have been removed.

Fortran 90 restricts the behavior of some features that were processor dependent in FORTRAN 77. Therefore, a program that conforms to the FORTRAN 77 standard and uses processor-dependent features can also conform to the Fortran 90 standard and yet behave differently than with some FORTRAN 77 systems. In the following situations, the Fortran 90 interpretation may be different from that of FORTRAN 77.

- Fortran 90 has more intrinsic functions than does FORTRAN 77, and it has a few intrinsic subroutines. Therefore, a standard-conforming FORTRAN 77 program can have a different interpretation under this standard if it invokes an external procedure that has the same name as one of the new standard intrinsic procedures, unless that procedure is specified in an `EXTERNAL` statement as recommended for nonintrinsic functions.

- If a named variable that is not in a common block is initialized in a `DATA` statement, it has the `SAVE` attribute in Fortran 90. In FORTRAN 77, if the value of the variable is changed or becomes undefined, its value on re-entry into a procedure is processor dependent. The Cray Research CF77 compiling system treated variables initialized in `DATA` statements as if they had appeared in a `SAVE` statement; the CF90 compiler continues this practice.

- In FORTRAN 77, an input list must never require more characters than are present in a record during formatted input. In Fortran 90, this restriction is removed for cases in which the `PAD=` specifier is `YES`; in this case, the input record is padded with as many blanks as necessary to satisfy the input item and the corresponding format. For more information on this, see the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.

- FORTRAN 77 permits a processor to supply extra precision for a real constant when it is used to initialize a `DOUBLE PRECISION` data object in a `DATA` statement. Fortran 90 does not permit this.

- The format of a floating-point zero written with a `G` edit descriptor is different in Fortran 90. The floating-point zero is written with an `E`*w*`.`*d* edit descriptor in FORTRAN 77, but it is written with an `F`*w*`.`*d* edit descriptor in the CF90 compiler. FORTRAN 77 output cannot be changed. Therefore, different compare files must be retained for FORTRAN 77 and Fortran 90 programs that use the `G` edit descriptor for floating-point output.

The Fortran 90 language standard

1.3

New data types, new operators, and new meanings for the existing operators and assignment provide ways for programmers to extend Fortran. These facilities allow programmers to create abstract data types by defining new types and the operations to be performed on them. Modules were introduced into Fortran as a convenient way to package these new data types and their operations. Modules can be used by the same user in different applications or can be distributed to several users on the same or different projects. This provides effective practical support for object-oriented programming, as well as enhancing both economy and efficiency .

The Fortran 90 standard (ISO/IEC 1539 : 1991) describes the syntax and semantics of the Fortran programming language. The standard addresses certain aspects of the Fortran processing system, but it does not address others. When specifications are not covered by the standard, the interpretation is processor dependent; that is, the processor defines the interpretation, but the interpretation for any two processors need not be the same. Typically , programs that rely on processor-dependent interpretations are not portable.

The standard specifies the following information:

- Syntax of Fortran statements and forms for Fortran programs
- Semantics of Fortran statements and Fortran programs
- Specifications for correct input data
- Appearance of standard output data

The standard does not specify the following information:

- The way in which each Fortran 90 compiler is written
- Operating system facilities defining the computing system
- Methods used to transfer data to and from peripheral storage devices and the nature of the peripheral devices
- Behavior of vendor extensions
- Size and complexity of a Fortran program and its data
- Hardware or firmware used to run the program
- The way values are represented and the way numeric values are computed
- Physical representation of data
- Characteristics of tapes, disks, and various storage media

Program conformance
1.3.1

A program conforms to the standard if all the statements are syntactically correct, execution of the program causes no violations of the standard (such as dividing by zero), and all the input data is in the correct form.

CF90 extensions to the Fortran 90 standard appear in notes throughout the text.

Processor conformance
1.3.2

In the Fortran 90 standard, the term *processor* means the combination of a Fortran compiler and the computing system that executes the code. A processor conforms to the standard if it compiles and executes programs that conform to the standard, provided that the Fortran program is not too large or complex for the computer system in question.

The CF90 compiler flags nonstandard usage when you specify `-e n` on the `f90` command line. This command-line option prints messages for CF90 extensions to the standard that are used in the program. As required by the standard, the CF90 compiler also flags the following items and provides the reason that the item is being flagged:

- Obsolescent features
- Kind type parameters not supported
- Violations of any syntax rules and the accompanying constraints
- Characters not permitted by the processor
- Illegal source form
- Violations of the scope rules for names, labels, operators, and assignment symbols

These conformance requirements were not present in previous Fortran standards.

The CF90 compiler includes extensions to the Fortran 90 standard. Because it processes programs according to the standard, it is considered to be a standard-conforming processor.

Portability
1.3.3

One of the main purposes of a standard is to describe how to write portable programs. However, some things are standard-conforming but not portable; for example, a program that computes a very large number like 10^{250}. Certain computing systems will not accommodate a number this large. Such a number could be part of a standard-conforming program, but it might not run on all systems; therefore, it might not be portable. Another example is a program that uses a deeper nesting of control constructs than is allowed by a particular compiler.

Fortran Concepts and Terms [2]

Terms are used in a precise way to describe a programming language, so this section introduces the fundamental terms needed to understand Fortran 90. You can consult the glossary in this manual for definitions of terms.

One of the major concepts involves the organization of a Fortran program. This topic is introduced in this section by presenting the high-level syntax rules for a Fortran program, which includes the principal constructs and statements that form a program. This section also describes the order in which constructs and statements must appear in a program and concludes with an example of a short Fortran 90 program.

Scope and association

2.1

In examining the basic concepts in Fortran, it helps to trace some of the important steps in its evolution. The results of the first few steps are familiar to Fortran programmers, but the later ones become relevant only when the new features of Fortran 90 are used.

The first version of Fortran produced in the late 1950s did not have user -defined subroutines or functions, but it did contain intrinsic functions.

Programmers soon realized the benefits of isolating definitive chunks of code into separate units known as function and subroutine subprograms. This not only provided a mechanism for structuring a program but permitted subprograms to be written once and then be called many times by the same program (or even be used by more than one program.) Equally important, subprograms could be compiled separately.

With this powerful tool came complications. For example, if both the main program and a subprogram use the variable named `X`, what is the connection between them? The answer is that, in general, there is no connection between `X` in the main program and `X` in a subprogram. Subprograms are separately compilable; an `X` in a different subprogram is not even known at compile time, so the simplest thing to do is have no connection between variables with the same name in different program units. Thus, if two different programmers work on different program units, neither one needs to worry about names picked by the other. This idea is described by saying that the two `X`s have different *scope*.

A subroutine could be written to do a summation of the first hundred integers and print the results. For example:

```
  SUBROUTINE TOTAL
  M = 0
  DO I = 1, 100
     M = M + I
  END DO
  WRITE(6, 9) M
9 FORMAT(I10)
  RETURN
  END
```

With this subroutine available, the main program could be written as follows:

```
CALL TOTAL
STOP
END
```

Suppose you decide that the subroutine would be more generally useful if it computed the sum but did not print it.

```
SUBROUTINE TOTAL
M = 0
DO I = 1, 100
   M = M + I
END DO
RETURN
END
```

A first attempt to use this subroutine might produce the following erroneous program:

```
      CALL TOTAL
      WRITE(6, 9) M
    9 FORMAT(I10)
      STOP
      END
```

This does not work because the variable `M` in the subroutine has nothing to do with variable `M` in the main program. A connection between the two values should exist, so when subroutines and functions were introduced, two schemes were provided to communicate values between them and other program units: procedure arguments and common blocks.

The following two complete programs communicate values in two ways. The first, program `ARGSUM`, uses a subroutine argument, and the second, program `COMSUM`, uses a common block to communicate values. The *names* in the different program units identify completely separate variables, yet their *values* are communicated from one to the other by using either arguments or common blocks, so the name of the variable holding the sum in the subroutine need not be the same as the corresponding variable in the calling program.

```
      PROGRAM ARGSUM
      CALL TOTAL(M)
      WRITE(6, 9) M
    9 FORMAT(I10)
      END

      SUBROUTINE TOTAL(ITOTAL)
      ITOTAL = 0
      DO I = 1, 100
         ITOTAL = ITOTAL + I
      END DO
      END
```

Program COMSUM, which follows, performs the same computation as program ARGSUM:

```
      PROGRAM COMSUM
      COMMON /CB/ M
      CALL TOTAL
      WRITE(6, 9) M
    9 FORMAT(I10)
      END

      SUBROUTINE TOTAL
      COMMON /CB/ ITOTAL
      ITOTAL = 0
      DO I = 1, 100
         ITOTAL = ITOTAL + I
      END DO
      END
```

To describe even these simple cases and appreciate how they work already requires the introduction of some terms and concepts. To precisely describe the phenomenon that the subroutine variable ITOTAL is not known outside the subroutine, the concept of *scope* is used. Because the scope of a variable in a common block is local, the scope of ITOTAL includes only the subroutine – not the main program.

The scope of ITOTAL is the subroutine; the scope of the variable M is the main program. However, the scope of the common block name CB is global. *Association* is used to describe the connection between M in the main program and ITOTAL in the subroutine. In the first example it is *argument association*, and in the second it is *storage association.*

To summarize, the *scope* of a variable is that part of the program in which it is known and can be used. Two variables may have the same name and nonoverlapping scopes; for example, there may be two completely different variables named X in two different subprograms. Association of variables means that there may be two different names for the same object; this permits sharing values under certain conditions.

With arguments available, it is natural to generalize the computation somewhat to allow the upper limit of the sum (100 in the example) to vary. Also, a function is more natural in this case than a subroutine, because the object of the computation is to return a single value. These changes produce the following program:

```
PROGRAM PTOTAL
INTEGER TOTAL
PRINT *, TOTAL(100)
END

FUNCTION TOTAL(N)
INTEGER TOTAL
TOTAL = 0
DO I = 1, N
   TOTAL = TOTAL + I
END DO
END
```

In this example, the scope of N is function TOTAL, but when function TOTAL is called from the main program in the PRINT statement, the value of N through argument association becomes 100. The scope of variable I is function TOTAL. The scope of function TOTAL is the whole program, but its type must be declared in the main program because by the implicit typing rules, TOTAL is not of type integer. Also note that there is a function named TOTAL, with global scope, and a variable named TOTAL is local to the function. The use of identifier TOTAL determines whether it is the local variable TOTAL or the global function name TOTAL. When TOTAL is used with an argument list, it is the function name; when used inside the function subprogram defining the function TOTAL, it is the local variable. Variable TOTAL computes and stores the value that is returned as the value of the function TOTAL.

It is possible to rewrite the example using internal procedures introduced in Fortran 90.

```
PROGRAM DO_TOTAL
  PRINT *, TOTAL(100)
  CONTAINS

  FUNCTION TOTAL(N)
     INTEGER TOTAL
     TOTAL = 0
     DO I = 1, N
        TOTAL = TOTAL + I
     END DO
  END FUNCTION TOTAL

END PROGRAM DO_TOTAL
```

This is similar to the previous example, but the function is placed prior to the `END` statement of the main program and the `CONTAINS` statement is inserted to mark the beginning of any internal functions or subroutines. In this case, the function `TOTAL` is not global but is local to the program `DO_TOTAL`. Also, the `FUNCTION` statement for `TOTAL` and the specifications that follow it specify `TOTAL` as a function of type integer with one integer argument `N`. The type of `TOTAL` must not be declared in the specification part of the program `DO_TOTAL`; to do so would create a duplicate declaration of `TOTAL`. The information about the type of the function and type of the argument is called the *interface* to the internal function.

To illustrate some other rules about scoping and association related to internal procedures, the example can be changed back to one that uses a subroutine, but one that is now internal.

```
PROGRAM DO_TOTAL
  INTEGER TOTAL
  CALL ADD_EM_UP(100)
  PRINT *, TOTAL
  CONTAINS

   SUBROUTINE ADD_EM_UP(N)
   TOTAL = 0
   DO I = 1, N
      TOTAL = TOTAL + I
   END DO
   END SUBROUTINE ADD_EM_UP

END PROGRAM DO_TOTAL
```

The preceding example shows that `TOTAL` in the internal subroutine and `TOTAL` in the main program are, indeed, the same variable. It does not need to be declared type integer in the subroutine. This is the result of *host association,* wherein internal procedures *inherit* information about variables from their host, which is the main program in this case. Variable `I` does not appear in the main program, so its scope is the internal subroutine.

Data declarations and procedures can be placed in a *module,* a new feature of Fortran 90. Then they can be used by other parts of the program. This scheme is illustrated using the summation function example again, as follows:

```
MODULE TOTAL_STUFF
CONTAINS
   FUNCTION TOTAL(N)
   INTEGER TOTAL, N, I
   TOTAL = 0
   DO I = 1, N
      TOTAL = TOTAL + I
   END DO
   END FUNCTION TOTAL
END MODULE TOTAL_STUFF

PROGRAM DO_TOTAL
USE TOTAL_STUFF
PRINT *, TOTAL(100)
END PROGRAM DO_TOTAL
```

The module and the program could be in different files. They can be compiled like subroutines, but unlike subroutines, the module must be available to the compiler when program `DO_TOTAL` is compiled.

The scope of variables `N` and `I` is function `TOTAL`; `N` gets its value 100 by argument association. The module name `TOTAL_STUFF` is global and any program can use the module, which causes the type and definition of function `TOTAL` to become available within that program. This is called *use association.*

When more extensive examples are constructed using such features as internal procedures within a procedure in a module, there is a need to have a deeper understanding of scope and association. These topics are introduced briefly in the following subsection and discussed in more detail in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

Scoping units
2.1.1

The scope of a program entity is that part of the entire executable program in which that entity is known, is available, and can be used. Some of the parts of a program that constitute the scope of entities are called *scoping units.* Scoping units range in extent of inclusiveness from parts of a statement to an entire program unit.

Some entities have scopes that are something other than a scoping unit. For example, the scope of a name, such as a variable name, can be global to an executing program. For more information on name scope and scoping units, see the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

Association
2.1.2

Association is the concept that is used to describe how different entities in the same program unit or different program units can share values and other properties. It is also a mechanism by which the scope of an entity is made larger. For example, argument association allows values to be shared between a procedure and the program that calls it. Storage association, set up by the use of `EQUIVALENCE` and `COMMON` statements, for example, allows two or more variables to share storage, therefore values, under certain circumstances. Use association and host association both allow entities described in one part of a program to be used in another part of the program. Use association makes entities defined in modules accessible, and host association makes entities in the containing environment available to an internal or module procedure. The complete description of all categories of association are described in the *CF90 Fortran Language Reference Manual* , *Volume 2*, publication SR–3903.

Program organization
2.2

A collection of program units constitutes an executable program. Program units can contain other smaller units.

Program units

2.2.1

A Fortran 90 program unit is one of the following:

- Main program
- External subprogram (subroutine or function)
- Module
- Block data

A Fortran program must contain one main program and can contain any number of the other types of program units. Program units define data environments and the steps necessary to perform calculations. Each program unit has an `END` statement to terminate the program unit. Each program unit also has an identifying initial statement, but the identifying statement for a main program is optional.

An external subprogram (a function or a subroutine) performs a task or calculation on entities available to the external subprogram. These entities can be the arguments to the subprogram that are provided in the reference, entities defined in the subprogram, or entities made accessible by other means, such as common blocks. A `CALL` statement invokes a subroutine. A function is invoked when its value is needed in an expression. The computational process that is specified by a function or subroutine subprogram is called a *procedure.* It may be invoked from other program units of the Fortran program. Neither module nor block data program units are executable, so they are not considered to be procedures. A main program also contains executable constructs, but it is not classified as a procedure.

A block data program unit contains only data definitions. It specifies initial values for a restricted set of data objects.

The program units described so far (main program, external subprogram, and block data) are familiar to users of FORTRAN 77. Fortran 90 introduces the module, which is a new kind of program unit.

A module contains public definitions that can be made accessible to other program units and subprograms. It also contains private definitions that are accessible only within the module. These definitions include data definitions, type definitions, definitions of subprograms known as *module subprograms,* and specifications of procedure interfaces. Module subprograms can be either subroutine or function subprograms. The procedures they define are called *module procedures*. Module subprograms can be invoked by other module subprograms in the module or by other program units that access the module.

A main program, external subprograms, and module subprograms may contain internal subprograms, which can be either subroutines or functions. The procedures they define are called *internal procedures.* Internal subprograms must not themselves contain internal subprograms, however. The main program, external subprogram, or module subprogram that contains an internal subprogram is referred to as the internal subprogram's *host.* Internal subprograms may be invoked by their host or by other internal subprograms in the same host.

A statement function also defines a procedure. Executable program units, module subprograms, and internal subprograms can all contain statement functions.

Figure 1 illustrates the organization of a sample Fortran program. Large arrows represent use association with the `USE` statement at the arrow tip. Small arrows represent subprogram references with the call at the arrow base.

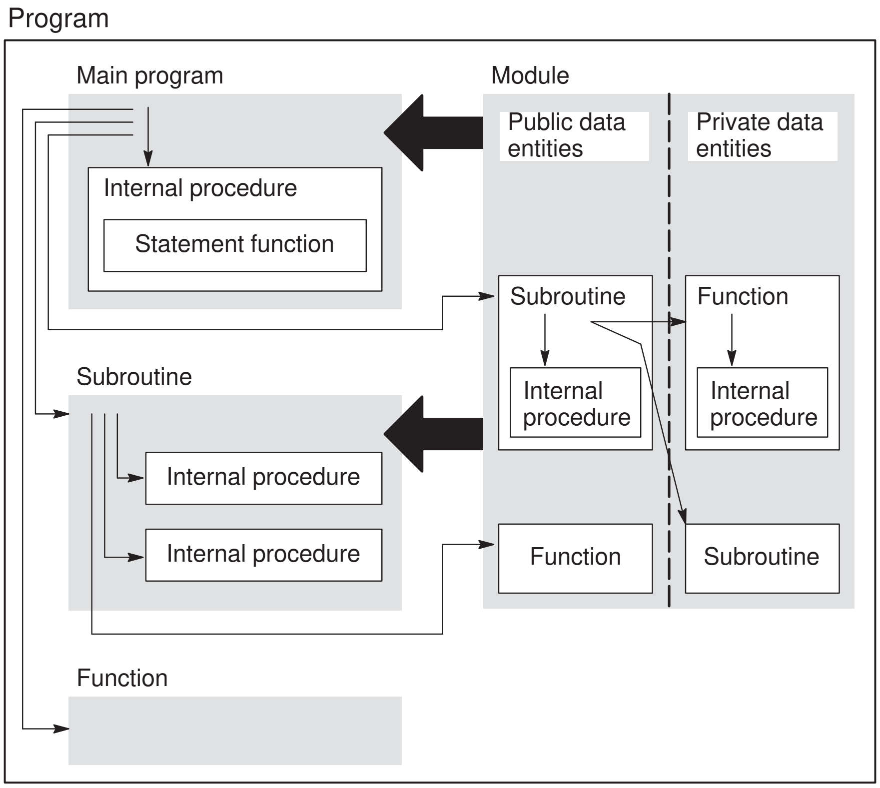

Figure 1. Program packaging example

Program units and subprograms are described more fully in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

Packaging

2.2.2

Opportunities for applying packaging concepts were limited in FORTRAN 77. An external subprogram might be thought of as a package, but it can contain only procedures, not data declarations that can be made available to other parts of a program. An entire FORTRAN 77 program can be thought of as a package made up of program units that consists of a main program, subroutine and function program units, and block data program units. In contrast, Fortran 90, with internal procedures and modules, provides more opportunities for packaging. This makes the packaging of a fair-sized program an important design consideration when planning a new Fortran application.

The most important benefit of packaging is the capability to hide information. Entities can be kept inaccessible except where they are actually needed. This provides some protection against inadvertent misuse or corruption, thereby improving program reliability. Packaging can make the logical structure of a program more apparent by hiding complex details at lower levels. Programs are therefore easier to comprehend and less costly to maintain. The Fortran 90 features that provide these benefits are internal procedures and modules.

As previously noted, internal procedures can appear in a main program, external subroutines, external functions, and module subprograms. They are known only within their host. The name of an internal procedure must not be passed as an argument, and an internal procedure must not itself be the host of another internal procedure. However, a statement function can appear within an internal procedure. Thus, in some ways, internal procedures are like external procedures and in other ways they are like statement functions.

Modules provide the most comprehensive opportunities to apply packaging concepts, as illustrated in Figure 1. In addition to several levels of organization and hiding, the entities specified in a module (types, data objects, procedures, interfaces, and so on) can be kept private to the module or made available to other scoping units by use association.

Data environment

2.3

Before a calculation can be performed, its data environment must be developed. The data environment consists of data objects that possess certain properties, attributes, and values. The steps in a computational process generally specify operations that are performed on operands (or objects) to create desired results or values. Operands can be constants, variables, constructors, or function references; each has a data type and (if defined) a value. In some cases the type can be assumed by the processor;

in other cases it may be declared. A data object has attributes other than type. Section 4 discusses data type in detail; section 5 discusses the other attributes of program entities; and sections 6 and 7 describe how data objects are used.

Data type
2.3.1

The Fortran 90 language provides five intrinsic data types: real, integer, complex, logical, and character, and it lets you define additional types. Sometimes it is natural to organize data in combinations consisting of more than one type. For example, assume that a program is written to monitor the patients in a hospital. For each patient, certain information must be maintained, such as the patient's name, room number, temperature, pulse rate, medication, and prognosis for recovery. Because all of this data describes one object (a particular patient), it would be convenient to have a means to refer to the aggregation of data by a single name. In Fortran 90, an aggregation of data values of different types is called a *structure*. To use a structure, a programmer must first define the type of the structure. After the new type is defined, any number of structures of that type can be declared. This mechanism might seem slightly cumbersome if only one such structure is needed in a program, but usually several are needed. The following is an example of a user-defined type with three components:

```
TYPE PATIENT
   INTEGER                PULSE_RATE
   REAL                   TEMPERATURE
   CHARACTER(LEN = 300)  PROGNOSIS
END TYPE PATIENT
```

After type PATIENT is defined, objects (structures) of the type can be declared. For example:

```
TYPE(PATIENT) JOHN_JONES, SALLY_SMITH
```

Kind
2.3.2

Some intrinsic types have more than one representation (or kind). Fortran 90 provides both single- and double-precision representations of the real and complex types; section 4 describes these data representations in detail. Fortran 90 permits more than one representation for the integer, logical, and character types. The CF90 compiler supports more than one representation for the integer and logical types, but it supports only a single character type (ASCII). Alternative representations for the integer type permit different ranges of integers.

Dimensionality
2.3.3

Single objects, whether intrinsic or user-defined, are scalar. Even though a structure has components, it is considered to be a scalar. A set of scalar objects, all of the same type, may be arranged in patterns involving columns, rows, planes, and higher-dimensioned configurations to form arrays. It is possible to have arrays of structures. An array may have a maximum of seven dimensions. The number of dimensions is called the *rank* of the array. It is declared when the array is declared and cannot change. The size of the array is the total number of elements and is equal to the product of the extents in each dimension. The shape of an array is determined by its rank and its extents in each dimension. Two arrays that have the same shape are said to be *conformable.* The following are examples of array declarations:

```
REAL COORDINATES(100, 100)
INTEGER DISTANCES(50)
TYPE(PATIENT) MATERNITY_WARD(20)
```

In Fortran 90, an array is treated as an object and is allowed to appear in an expression or be returned as a function result. Intrinsic operations involving arrays of the same shape are performed element-by-element to produce an array result of the same shape. There is no implied order in which the element-by-element operations are performed.

A portion of an array, such as an element or section, can be referenced as a data object. An array *element* is a single element of the array and is scalar. An array *section* is a subset of the elements of the array and is itself an array .

Dynamic data
2.3.4

There are three sorts of dynamic data objects in Fortran 90: pointers, allocatable arrays, and automatic data objects.

Data objects in Fortran 90 can be declared to have the POINTER attribute. Pointer objects must be associated with a target before they can be used in any calculation. This is accomplished by allocation of the space for the target or by assignment of the pointer to an existing target. The association of a pointer with a target can change dynamically as a program is executed. If the pointer object is an array, its size and shape could change dynamically , but its rank is fixed by its declaration. The following is an example of pointer array declaration and allocation:

```
REAL, POINTER :: LENGTHS(:)
ALLOCATE(LENGTHS (200))
```

Following execution of an ALLOCATE statement, LENGTHS points at (is associated with) an array with 200 elements. The elements as yet have no values.

An array can be declared to have the ALLOCATABLE attribute. Space must be allocated for the array before it can be used in any calculation. The array can be deallocated and reallocated with a different size as the program executes. The size and shape can change at each allocation, but the rank is fixed by the declaration. The following is an example of allocatable array declaration and allocation:

```
REAL, ALLOCATABLE :: LENGTHS(:)
ALLOCATE (LENGTHS(200))
```

The similarities of these examples reflect the similarity of some of the uses of allocatable arrays and pointers, but pointers have more functionality. Pointers can be used to create dynamic data structures, such as linked lists and trees. The target of a pointer can be changed by reallocation or pointer assignment. The extents of an allocatable array can be changed only by deallocating and reallocating the array. If the values of the elements of an allocatable array are to be preserved, a new array must be allocated and the values moved to the new array before the old array is deallocated.

An automatic data object can be an array or a character string. Automatic objects can be declared in a subprogram. These local data objects are created on entry to the subprogram and disappear when the execution of the subprogram completes. They are useful in subprograms for temporary arrays and characters strings whose sizes are different for each reference to the subprogram. The following is an example of a subprogram unit with automatic array TEMP:

```
SUBROUTINE SWAP_ARRAYS(A, B)
   REAL, DIMENSION(:) :: A, B
   REAL, DIMENSION(SIZE(A)) :: TEMP

   TEMP = A
   A = B
   B = TEMP
END SUBROUTINE SWAP_ARRAYS
```

`A` and `B` are assumed-shape array arguments; that is, they take on the shape of the actual argument. `TEMP` is an automatic array that is created the same size as `A` on entry to subroutine `SWAP_ARRAYS`. `SIZE` is an intrinsic function that is permitted in a declaration statement.

Program execution

2.4

During program execution, constructs and statements are executed in a prescribed order. Variables become defined with values and can be redefined later in the execution sequence. Procedures are invoked, perhaps recursively . Space can be allocated and later deallocated. Pointers can change their targets.

Execution sequence

2.4.1

Program execution begins with the first executable construct in the main program. An executable construct is an instruction to perform one or more of the computational actions that determine the behavior of the program or control the flow of the execution of the program. It can perform arithmetic, compare values, branch to another construct or statement in the program, invoke a procedure, or read from or write to a file or device. When a procedure is invoked, its execution begins with the first executable construct after the entry point in the procedure. On normal return from a procedure invocation, execution continues where it left off. The following are examples of executable statements:

```
     READ(5, *) Z, Y
     X = (4.0 * Z) + BASE
     IF (X > Y) GO TO 100
     CALL CALCULATE(X)
100  Y = Y + 1
```

Unless a branching statement or control construct is encountered, executable statements are executed in the order in which they appear in a program unit until a STOP, RETURN, or END statement is executed. Branch statements specify a change in the execution sequence and consist of the various forms of GO TO statements, a procedure reference with alternative return specifiers, and input/output (I/O) statements with branch label specifiers such as ERR=, END=, and EOR= specifiers. The control constructs (IF, CASE, and DO) can cause internal branching implicitly within the structure of the construct. Section 8, which begins on page 323, discusses control flow within a program in detail.

Section 2

Definition and undefinition

2.4.2

Most variables have no value when execution begins; they are considered to be *undefined.* Exceptions are variables that are initialized in DATA statements or type declaration statements; these are considered to be *defined.* A variable can acquire a value or change its current value, typically by the execution of an assignment statement or an input statement. Thus it can assume different values at different times, and under some circumstances it can become undefined. *Defined* and *undefined* are the Fortran terms that are used to specify the definition status of a variable. The *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903, describes the events that cause variables to become defined and undefined.

A variable is considered to be defined only if all parts of it are defined. For example, all elements of an array, all components of a structure, or all characters of a character string must be defined; otherwise, the array, structure, or string is undefined. Fortran 90 permits zero-sized arrays and zero-length strings; these are always considered to be defined.

Pointers have both a definition status and an association status. When execution begins, the association status of all pointers is undefined. During execution a pointer may become nullified by the execution of a NULLIFY statement, in which case its association status becomes *disassociated,* or it may become associated with a target by the execution of an ALLOCATE or pointer assignment statement, in which case its association status becomes *associated.* Even when the association status of a pointer is defined, the pointer is not considered to be defined unless the target with which it is associated is defined. Pointer targets become defined in the same way that other variables becomes defined, typically by the execution of an assignment or input statement. When an allocatable array is allocated by the execution of an `ALLOCATE` statement, it is undefined until some other action occurs that causes it to become defined with values for all array elements.

Dynamic behavior
2.4.3

Fortran 90 introduces the following new types of dynamic behavior:

- Recursion
- Allocation and deallocation
- Pointer assignment

Many algorithms can be expressed with the use of *recursion* , which occurs when a subroutine or function references itself, either directly or indirectly. The keyword `RECURSIVE` must be present in the `SUBROUTINE` or `FUNCTION` statement if the procedure is referenced recursively. Recursive subroutines and functions are described in the *CF90 Fortran Language Reference Manual* , *Volume 2*, publication SR–3903.

No space exists for a pointer until the pointer is allocated or associated with an existing target. No space exists for an allocatable array until the array is allocated. The rank of array pointers and allocatable arrays is fixed by declaration, but the extents in each dimension (and thus the size of the arrays) is determined during execution by calculation or from input values.

The `ALLOCATE` and `DEALLOCATE` statements give Fortran programmers mechanisms to configure objects to the appropriate shape. Only pointers and allocatable arrays can be allocated. It is not possible to deallocate an object unless it was previously allocated, and it is not possible to deallocate a part of an object unless it is a pointer component of a structure. It is possible to inquire whether a pointer is currently associated and whether an allocatable array is currently allocated. Section 5, which begins on page 131, describes the declaration of pointers and allocatable arrays. Section 6, which begins on page 203, describes the `ALLOCATE` and `DEALLOCATE` statements. The *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903, describes the `ASSOCIATED` intrinsic inquiry function for pointers and the `ALLOCATED` intrinsic inquiry function for allocatable arrays.

Pointers are more flexible than allocatable arrays, but they are also more complicated and less efficient. A pointer need not be an array; it can be a scalar of any type. A pointer need not be associated with allocated space; any object with the TARGET attribute can become a pointer target. A pointer assignment statement is provided to associate a pointer with a target (declared or allocated). It makes use of the symbol => rather than the single character =. In all other respects, pointer assignment statements are executed in the same way that ordinary assignment statements are executed, except that instead of assigning a value they associate a pointer with a target, as is shown in the following example:

```
REAL, TARGET :: VECTOR(100)
REAL, POINTER :: ODDS(:)
   . . .
ODDS => VECTOR(1:100:2)
```

The pointer assignment statement associates ODDS with the odd elements of VECTOR. The following assignment statement defines each odd element of VECTOR with the value 1.5:

```
ODDS = 1.5
```

Later in the execution sequence, pointer ODDS could become associated with a different target by pointer assignment or allocation, as long as the target is a one-dimensional, real array. Section 7, which begins on page 233, describes the pointer assignment statement.

Summary of forms

2.5

This subsection shows the forms of the higher-level components of a Fortran 90 program. The notation used in the forms is the same as that used to show the syntax forms in all the remaining subsections of this manual. The complete Backus-Naur form (BNF), as given in the Fortran 90 standard, is included in the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.

Program units
2.5.1

The Fortran 90 standard rule R202 defines a *program_unit* as follows:

R201	*executable_program*	**is**	*program_unit* [*program_unit*] ...
R202	*program_unit*	**is**	*main_program*
		or	*external_subprogram*
		or	*module*
		or	*block_data*
R1101	*main_program*	**is**	[*program_stmt*] [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_program_stmt*
R203	*external_subprogram*	**is**	*function_subprogram*
		or	*subroutine_subprogram*
R1215	*function_subprogram*	**is**	*function_stmt* [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_function_stmt*
R1219	*subroutine_subprogram*	**is**	*subroutine_stmt* [*specification_part*] [*execution_part*] [*internal_subprogram_part*] *end_subroutine_stmt*
R1104	*module*	**is**	*module_stmt* [*specification_part*] [*module_subprogram_part*] *end_module_stmt*

R1110	*block_data*	**is**	*block_data_stmt*
			[*specification_part*]
			end_block_data_stmt
R204	*specification_part*	**is**	[*use_stmt*] ...
			[*implicit_part*]
			[*declaration_construct*] ...
R205	*implicit_part*	**is**	[*implicit_part_stmt*] ...
			implicit_stmt
R206	*implicit_part_stmt*	**is**	*implicit_stmt*
		or	*parameter_stmt*
		or	*format_stmt*
		or	*entry_stmt*
R207	*declaration_construct*	**is**	*derived_type_def*
		or	*interface_block*
		or	*type_declaration_stmt*
		or	*specification_stmt*
		or	*parameter_stmt*
		or	*format_stmt*
		or	*entry_stmt*
		or	*stmt_function_stmt*
R208	*execution_part*	**is**	*executable_construct*
			[*execution_part_construct*] ...
R209	*execution_part_construct*	**is**	*executable_construct*
		or	*format_stmt*
		or	*data_stmt*
		or	*entry_stmt*

R210	*internal_subprogram_part*	**is**	*contains_stmt*
			internal_subprogram
			[*internal_subprogram*] ...
R211	*internal_subprogram*	**is**	*function_subprogram*
		or	*subroutine_subprogram*
R212	*module_subprogram_part*	**is**	*contains_stmt*
			module_subprogram
			[*module_subprogram*] ...
R213	*module_subprogram*	**is**	*function_subprogram*
		or	*subroutine_subprogram*
R214	*specification_stmt*	**is**	*access_stmt*
		or	*allocatable_stmt*
		or	*common_stmt*
		or	*data_stmt*
		or	*dimension_stmt*
		or	*equivalence_stmt*
		or	*external_stmt*
		or	*intent_stmt*
		or	*intrinsic_stmt*
		or	*namelist_stmt*
		or	*optional_stmt*
		or	*pointer_stmt*
		or	*save_stmt*
		or	*target_stmt*
R215	*executable_construct*	**is**	*action_stmt*
		or	*case_construct*
		or	*do_construct*
		or	*if_construct*
		or	*where_construct*

R216	*action_stmt*	**is**	*allocate_stmt*
		or	*assignment_stmt*
		or	*backspace_stmt*
		or	*call_stmt*
		or	*close_stmt*
		or	*computed_goto_stmt*
		or	*continue_stmt*
		or	*cycle_stmt*
		or	*deallocate_stmt*
		or	*endfile_stmt*
		or	*end_function_stmt*
		or	*end_program_stmt*
		or	*end_subroutine_stmt*
		or	*exit_stmt*
		or	*goto_stmt*
		or	*if_stmt*
		or	*inquire_stmt*
		or	*nullify_stmt*
		or	*open_stmt*
		or	*pointer_assignment_stmt*
		or	*print_stmt*
		or	*read_stmt*
		or	*return_stmt*
		or	*rewind_stmt*
		or	*stop_stmt*
		or	*where_stmt*
		or	*write_stmt*
		or	*arithmetic_if_stmt*
		or	*assign_stmt*
		or	*assigned_goto_stmt*
		or	*pause_stmt*

Main program
2.5.2

R1101 defines a main program. Typically, a main program takes the following form:

```
[ PROGRAM program_name ]
   [ specification_part ]
   [ execution_part ]
   [ CONTAINS
   internal_subprogram
   [ internal_subprogram ] ... ]
END [PROGRAM[program_name ]]
```

External subprogram
2.5.3

R203 defines external subprograms. Typically, an external subprogram takes one of the following forms:

```
function_stmt
   [ specification_part ]
   [ execution_part ]
   [ CONTAINS
   internal_subprogram
   [ internal_subprogram ] ... ]
end_function_stmt
```

```
subroutine_stmt
   [ specification_part ]
   [ execution_part ]
   [ CONTAINS
   internal_subprogram
   [internal_subprogram ] ... ]
end_subroutine stmt
```

Module
2.5.4

R1104 defines modules. Typically, a module takes the following form:

```
MODULE module_name
   [ specification_part ]
   [ CONTAINS
   module_subprogram
   [ module_subprogram ] ... ]
END [ MODULE [ module_name ]
```

Block data
2.5.5

R1110 defines block data program units. Typically, a block data program unit takes the following form:

```
BLOCK DATA [ block_data_name ]
   [ specification_part ]
END [ BLOCK DATA [ block_data_name ] ]
```

Internal subprogram
2.5.6

R211 defines internal subprograms. Typically, an internal subprogram takes one of the following forms:

function_stmt   [*specification_part*]   [*executable_part*] *end_function_stmt*

subroutine_stmt   [*specification_part*]   [*executable_part*] *end_subroutine stmt*

Procedure headings
2.5.7

R1216 and R1220 define `FUNCTION` and `SUBROUTINE` statements as follows:

[`RECURSIVE`] [*type_spec*] `FUNCTION` *function_name* ≡   ≡ ([*dummy_argument_list*]) [`RESULT` (*result_name*)]

[`RECURSIVE`] `SUBROUTINE` *subroutine_name* ≡   ≡ [([*dummy_argument_list*])]

Section 2

Procedure endings

2.5.8

R1218 and R1222 define the END FUNCTION and END SUBROUTINE statement, which are as follows:

```
END [ FUNCTION [ function_name ] ]
```

```
END [ SUBROUTINE [ subroutine_name ] ]
```

Specification constructs

2.5.9

A specification part generally can contain any of the following. Some program units or subprograms may not allow some of these specification constructs. See the descriptions for each program unit or subprogram description for restrictions.

- USE statements
- PARAMETER statements
- FORMAT statements
- ENTRY statements
- Derived type definitions
- Interface blocks
- Type declaration statements
- Other specification statements

Derived-type definition
2.5.10

R422 defines derived-type definitions. Typically, derived-type definitions take the following form:

```
TYPE [ [ , access_spec ] :: ] type_name
  [ PRIVATE ]
  [ SEQUENCE ]
  type_spec [ [ , component_attr_spec_list ] :: ] component_decl_list ...
  [ type_spec [ [ , component_attr_spec_list ] :: ] component_decl_list ... ] ...
END TYPE [ type_name ]
```

Interface block
2.5.11

R1201 defines an interface block. Typically, an interface block takes the following form:

```
INTERFACE [ generic_spec ]
  [ interface_body ] ...
  [ MODULE PROCEDURE procedure_name_list ] ...
END INTERFACE
```

Specification statements
2.5.12

The specification statements are shown in the following box. See the subsections in which they are described for more information on their forms and syntax.

Section 2

```
ALLOCATABLE [ :: ] allocatable_array_list
COMMON [ / [ common_block_name ] / ] common_block_object_list
DATA data_statement_object_list / data_statement_value_list /
DIMENSION array_dimension_list
EQUIVALENCE equivalence_set_list
EXTERNAL external_name_list
FORMAT ( [ format_item_list ] )
IMPLICIT implicit_spec
INTENT ( intent_spec ) [ :: ] dummy_argument_name_list
INTRINSIC intrinsic_procedure_name_list
NAMELIST / namelist_group_name / namelist_group_object_list
OPTIONAL [ :: ] optional_object_list
PARAMETER ( named_constant_definition_list )
POINTER [ :: ] pointer_name_list
PUBLIC [ [ :: ] module_entity_name_list ]
PRIVATE [ [ :: ] module_entity_name_list ]
SAVE [ [ :: ] saved_object_list ]
TARGET [ :: ] target_name_list
USE module_name [ , rename_list ]
USE module_name , ONLY : [ access_list ]
type_spec [ [ , attr_spec ] ... :: ] object_declaration_list
```

Type declaration statements

2.5.13

R502 defines the type specification statements. Typically , the type declaration statements take the following form:

INTEGER [([KIND=] *kind_parameter*)]
REAL [([KIND=] *kind_parameter*)]
DOUBLE PRECISION
COMPLEX [([KIND=] *kind_parameter*)]
CHARACTER [([LEN=] *length_parameter*)]
CHARACTER ([LEN=] *length_parameter* [, [KIND=] *kind_parameter*)
CHARACTER (KIND= *kind_parameter* [, LEN= *length_parameter*])
CHARACTER * *char_length*
LOGICAL [([KIND=] *kind_parameter*)]
TYPE (*type_name*)

Attribute specifications
2.5.14

R503 defines the attribute specifications. Subsection 2.5.12, page 37, introduced *attr_spec* . Typically , attribute specifications take the following form:

ALLOCATABLE
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
TARGET

Execution part

2.5.15

An execution part (R208) can contain the following:

- *action_statement*
- *case_construct*
- *do_construct*
- *if_construct*
- *where_construct*

Action statements

2.5.16

R216 defines action statements. See the subsections in which they are described for more information on their forms and syntax. Typically, action statements take the following form:

```
ALLOCATE ( allocation_list [ , STAT= scalar_integer_variable ] )
ASSIGN label TO scalar_integer_variable
BACKSPACE external_file_unit
BACKSPACE ( position_spec_list )
BUFFER IN ( id , mode ) ( start_loc , end_loc )
BUFFER OUT ( id , mode ) ( start_loc , end_loc )
CALL subroutine_name [ ( [ actual_argument_spec_list ] ) ]
CLOSE ( close_spec_list )
CONTINUE
CYCLE [ do_construct_name ]
DEALLOCATE ( name_list [ , STAT= scalar_integer_variable ] )
ENDFILE external_file_unit
ENDFILE ( position_spec_list )
EXIT [ do_construct_name ]
GO TO label
GO TO ( label_list ) [ , ] scalar_integer_expression
GO TO scalar_integer_variable [ [ , ] ( label_list ) ]
IF ( scalar_logical_expression ) action_statement
IF ( scalar_numeric_expression ) label , label , label
INQUIRE ( inquire_spec_list ) [ output_item_list ]
NULLIFY ( pointer_object_list )
OPEN ( connect_spec_list )
PAUSE [ access_code ]
PRINT format [ , output_item_list ]
READ ( io_control_spec_list ) [ input_item_list ]
READ format [ , input_item_list ]
RETURN [ scalar_integer_expression ]
REWIND external_file_unit
REWIND ( position_spec_list )
STOP [ access_code ]
WHERE ( array_logical_expression ) array_assignment_statement
WRITE ( io_control_spec_list ) [ output_item_list ]
pointer_variable => target_expression
variable = expression
```

Section 2

CASE construct
2.5.17

R808 defines the CASE construct. Typically, CASE constructs take the following form:

```
SELECT CASE ( case_expr )
   [ CASE case_selector
     block ] ...
   [ CASE DEFAULT
     block ]
END SELECT
```

DO construct
2.5.18

R816 defines DO constructs. Typically, DO constructs take the following form:

```
DO [ label ]
     block
end_do
```

```
DO [ label ] [ , ] do_variable = scalar_numeric_expr , ≡
≡ scalar_numeric_expr [, scalar_numeric_expr ]
     block
end_do
```

```
DO [ label ] [ , ] WHILE ( scalar_logical_expr )
     block
end_do
```

Section 2

`IF` construct

2.5.19

R802 defines the `IF` construct. Typically, `IF` constructs take the following form:

```
IF ( scalar_logical_expr ) THEN
    block
  [ ELSE IF ( scalar_logical_expr ) THEN
    block ] ...
  [ ELSE
    block ]
END IF
```

`WHERE` construct

2.5.20

R739 defines the `WHERE` construct. Typically, `WHERE` constructs take the following form:

```
WHERE ( mask_expr )
    [ assignment_stmt ] ...
  [ ELSEWHERE
    [ assignment_stmt ] ... ]
END WHERE
```

Ordering requirements

2.6

Within program units and subprograms, there are ordering requirements for statements and constructs. The syntax rules in the previous subsection do not fully describe the ordering requirements. They are illustrated in both Figure 2, page 44, and Figure 3, page 45. Generally, data declarations and specifications must precede executable constructs and statements, although FORMAT, DATA, NAMELIST, and ENTRY statements can appear among the executable statements. USE statements, if any, must appear first. Internal or module subprograms, if any, must appear last following a CONTAINS statement.

ANSI/ISO: The Fortran 90 standard does not allow NAMELIST statements to be intermixed with executable statements and constructs. The CF90 compiler eases this restriction.

In Figure 2, a vertical line separates statements and constructs that can be interspersed; a horizontal line separates statements that must not be interspersed.

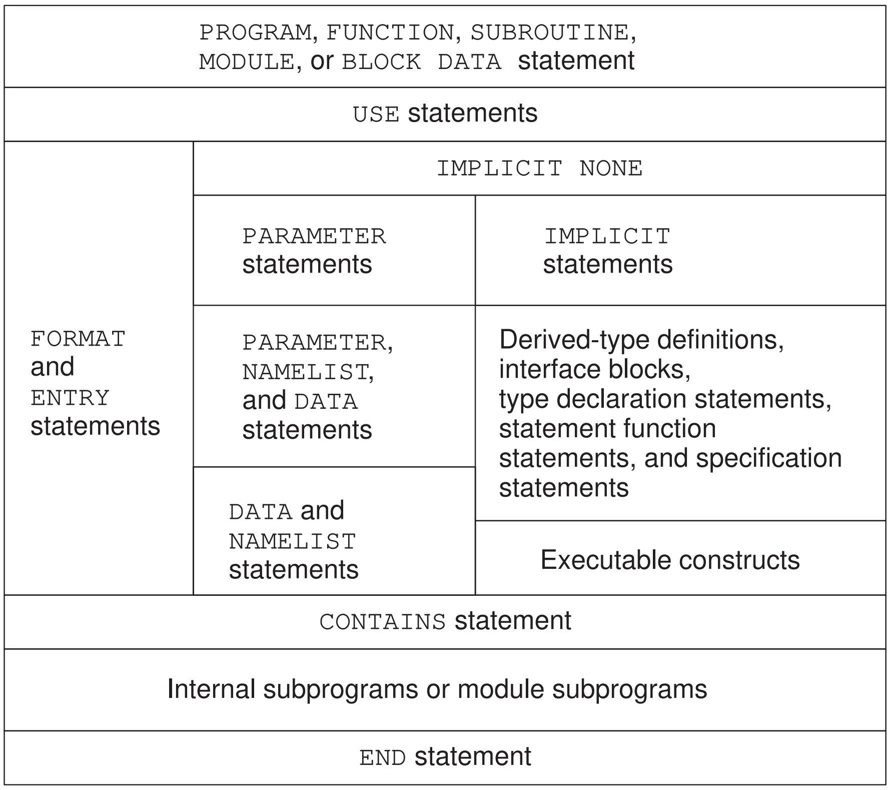

Figure 2. Requirements on statement ordering

There are restrictions on the places where some statements may appear. Figure 3 summarizes these restrictions.

Scoping unit	Main program	Module	Block data	External subprog	Module subprog	Internal subprog	Interface body
`USE` statement	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`ENTRY` statement	No	No	No	Yes	Yes	No	No
`FORMAT` statement	Yes	No	No	Yes	Yes	Yes	No
Misc. declarations (see note)	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`DATA` statement	Yes	Yes	Yes	Yes	Yes	Yes	No
Derived-type definition	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Interface block	Yes	Yes	No	Yes	Yes	Yes	Yes
Statement function	Yes	No	No	Yes	Yes	Yes	No
Executable statement	Yes	No	No	Yes	Yes	Yes	No
`CONTAINS`	Yes	Yes	No	Yes	Yes	No	No

Note: Miscellaneous declarations are `PARAMETER` statements, `IMPLICIT` statements, type declaration statements, and specification statements.

Figure 3. Restrictions on the appearance of statements

Section 2

Example Fortran 90 program

2.7

The following simple Fortran 90 program consists of one program unit, the main program. Three data objects are declared: `H`, `T`, and `U`. These become the loop indexes in a triply-nested loop structure that contains an `IF` statement that conditionally executes an I/O statement.

```
PROGRAM SUM_OF_CUBES
! This program prints all 3-digit numbers that
! equal the sum of the cubes of their digits.
INTEGER :: H, T, U
DO H = 1, 9
  DO T = 0, 9
    DO U = 0, 9
    IF (100*H + 10*T + U == H**3 + T**3 + U**3) &
      PRINT "(3I1)", H, T, U
    END DO
  END DO
END DO
END PROGRAM SUM_OF_CUBES
```

This Fortran 90 program produces the following output:

```
153
370
371
407
```

Language Elements and Source Form [3]

This section describes the language elements that a Fortran statement can contain. Language elements consist of lexical tokens, which include names, keywords, operators, and statement labels. Rules for forming lexical tokens from the characters in the Fortran character set are also presented.

The source form describes how to place these elements on a line in a Fortran program. There are two source forms in Fortran 90. One form is oriented toward the Hollerith punched card common in the 1960s and is restricted to 80 positions. It is called *fixed* source form. The other is new in Fortran 90 and is oriented toward terminal input of source code. It is called *free* source form.

The `INCLUDE` line is a new feature in Fortran 90 that allows you to include source code from a specified file. The CF77 compiling system, which is supported on Cray PVP systems and CRAY T3D systems, supported `INCLUDE` lines in its 6.0 release. It is a convenient way to place the same text in several places in a program.

Section 3

CF90 character set

3.1

The CF90 character set contains the following characters:

- The Fortran 90 standard character set, which includes the 26 uppercase letters of the alphabet, the corresponding 26 lowercase letters, and several special characters.
- The numerical digits `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`.
- The underscore character (_).
- The newline and tab characters, which are control characters and have no graphic representation. These characters are part of the processor-dependent control characters as allowed by the standard.

Table 1 shows the special characters of the CF90 character set.

Table 1. CF90 special characters

Graphic	Character name	Graphic	Character name
	Blank	:	Colon
=	Equals	!	Exclamation point
+	Plus	"	Quotation mark
−	Minus	%	Percent
*	Asterisk	&	Ampersand
/	Slash	;	Semicolon
(	Left parenthesis	<	Less than
)	Right parenthesis	>	Great than
,	Comma	?	Question mark
'	Apostrophe	$	Currency symbol
@	At sign†	.	Decimal point or period

ANSI/ISO: The at sign (`@`) is not included in the Fortran 90 standard character set. The newline and tab characters are not part of the graphic character set as defined by the standard, but the standard allows them as part of the processor-dependent control character set.

Note: The CF90 compiler does not support use of the at sign (`@`) on SPARC systems.

The 52 letters define the syntax class *letter*.

† Use of the at sign (`@`) differs depending on your hardware platform. On Cray PVP systems and CRAY T3E systems, the CF90 compiler allows you to use the at sign (`@`) in certain cases; these cases are described in subsection 3.2.2, page 52. The at sign (`@`) is not supported as a CF90 compiler character on SPARC systems.

In most cases, the CF90 compiler is not case sensitive; lowercase letters are considered the same as uppercase letters. In the following cases, however, lowercase and uppercase letters are considered to have different data values:

- Within a character constant
- Within a quotation mark, apostrophe, or H edit descriptor
- Within a Hollerith constant (nonstandard)

ANSI/ISO: Hollerith data is not included in the Fortran 90 standard.

The CF90 compiler considers the following two statements to be equivalent:

```
PRINT *, N
Print *, n
```

The CF90 compiler distinguishes between uppercase and lowercase letters in the FILE= or NAME= specifier in an OPEN or an INQUIRE statement.

The digits 0 through 9 define the syntax class *digit* . The digits are assumed to be decimal numbers when used to describe a numeric value, except in binary , octal, and hexadecimal (BOZ) literal constants or input/output (I/O) records corresponding to B, O, or Z edit descriptors.

For example, in the following DATA statement, the digits of the first constant are decimal digits, those of the second constant are binary digits, and those of the third are hexadecimal digits:

```
DATA  X, I, J / 4.89, B'1011', Z'BAC91' /
```

The underscore can be used to make names more readable. For example, in the identifier NUMBER_OF_CARS, each underscore is used to separate the obvious English words. It is a significant character in any name. An underscore cannot be used as the first character of a name, but it can be the last character. An underscore is also used to separate the kind value from the actual value of a literal constant (for example, 123_2).

There are 22 special characters used for operators like multiply and add. They are also used as separators or delimiters in Fortran statements. Separators and delimiters make the form of a statement unambiguous.

Section 3

ANSI/ISO: On Cray PVP systems and CRAY T3E systems, the at sign (`@`) can be used in an identifier in the same way that you would use an underscore (_). The at sign cannot be used as the first character in an identifier. Using the at sign is not recommended because Cray Research reserves it for internal use.

Note: The CF90 compiler does not support use of the at sign (`@`) on SPARC systems.

Fortran's treatment of uppercase and lowercase letters can lead to portability problems when calling subprograms written in other languages. The problem occurs because the standard does not specify the case of letters used for external names. The following program fragment illustrates the problem:

```
EXTERNAL FOO
   . . .
CALL FOO
   . . .
END
```

The CF90 compiler uses uppercase for external names. Thus, the CF90 compiler uses `FOO` as the external name. If the subprogram were written in C, which is case sensitive, and if `foo` were written in lowercase, the external name used in C would then be different from the name produced by the CF90 compiler.

The `NAME` compiler directive allows you to specify a case-sensitive external name in a Fortran program. You can use this directive, for example, when writing calls to C routines. The case-sensitive external name is specified on the `NAME` directive. For more information on the `NAME` directive, see the *CF90 Commands and Directives Reference Manual*, publication SR–3901.

Lexical tokens

3.2

A statement is constructed from low-level syntax. The low-level syntax describes the basic language elements, called *lexical tokens,* in a statement. A lexical token is the smallest meaningful unit of a statement and can consist of 1 or more characters. Tokens are names, keywords, literal constants (except for complex literal constants), labels, operator symbols, comma, =, =>, `:`, `::`, `;`, `%`, and delimiters. A literal of type complex consists of several tokens. Examples of operator symbols are + and `//`.

Delimiters are pairs of symbols that enclose parts of a statement. The following symbol pairs are delimiters:

```
/ ... /
( ... )
(/ ... /)
```

In the following statements, the slashes distinguish the value list from the object list in a `DATA` statement, the parentheses are delimiters that mark the beginning and end of the argument list in the `CALL` statement, and the pairs (/ and /) mark the beginning and end of the elements of an array constructor:

```
DATA  X, Y / 1.0, -10.2/
CALL PRINT_LIST(LIST, SIZE)
VECTOR = (/ 10, 20, 30, 40 /)
```

Statement keywords

3.2.1

Statement keywords appear in uppercase letters in the syntax rules. Some statement keywords identify the statement, such as in the following `DO` statement:

```
DO I = 1, 10
```

`DO` is a statement keyword that identifies the `DO` statement. Other keywords identify parts of a statement such as `ONLY` in a `USE` statement or `WHILE` in one of the forms of a `DO` construct, as follows:

```
DO WHILE( .NOT. FOUND )
```

Section 3

Others specify options in the statement such as `IN`, `OUT`, or `INOUT` in the `INTENT` statement.

There are three statements in Fortran 90 that have no statement keyword. They are the assignment statement, the pointer assignment statement, and the statement function.

Some sequences of capital letters in the formal syntax rules are not statement keywords. For example, `EQ`, in the lexical token `.EQ.`, and `EN`, as an edit descriptor, are not statement keywords.

A dummy argument keyword, a different sort of keyword, is discussed in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

Section 3

Names
3.2.2

Variables, named constants, program units, common blocks, procedures, arguments, constructs, derived types (types for structures), namelist groups, structure components, dummy arguments, and function results are among the elements in a program that have a name. Fortran 90 permits up to 31 characters in a name.

R301	*character*	**is**	*alphanumeric_character*
		or	*special_character*
R302	*alphanumeric_character*	**is**	*letter*
		or	*digit*
		or	*underscore*
CRI		**or**	*currency_symbol*
CRI		**or**	*at_sign*
R303	*underscore*	**is**	_
CRI	*currency_symbol*	**is**	$
CRI	*at_sign*	**is**	@
R304	*name*	**is**	*letter* [*alphanumeric_character*] ...

A name must begin with a letter and can consist of letters, digits, and underscores. The CF90 compiler allows you to use the underscore (_), at sign (`@`), and currency symbol in a name, but none of these can be the first character of a name. Use of `@` and `$` is not recommended, however, because these are reserved for internal use by Cray Research, Inc.

Following are examples of names:

```
A
CAR_STOCK_NUMBER
A__BUTTERFLY
Z_28
TEMP_
```

ANSI/ISO: The Fortran 90 standard does not allow the currency symbol (`$`) or the at sign (`@`) character in a name. The at sign (`@`) is not included in the Fortran 90 standard character set.

Note: The CF90 compiler does not support use of the at sign (`@`) on SPARC systems.

Section 3

Constants
3.2.3

A *constant* is a syntactic notation for a value. The value can be of any intrinsic type; that is, it can be a numeric (integer, real, or complex) value, a character value, or a logical value. A *constant* is defined as follows:

R305	*constant*	**is**	*literal_constant*
		or	*named_constant*
R306	*literal_constant*	**is**	*int_literal_constant*
		or	*real_literal_constant*
		or	*complex_literal_constant*
		or	*logical_literal_constant*
		or	*char_literal_constant*
		or	*boz_literal_constant*
CRI		**or**	*typeless_constant*

CRI	*typeless_constant*	**is**	*octal_typeless_constant*
		or	*hexadecimal_typeless_constant*
		or	*binary_typeless_constant*
CRI	*octal_typeless_constant*	**is**	*digit* [*digit* ...] `B`
		or	`O"` *digit* [*digit* ...] `"`
		or	`O'` *digit* [*digit* ...] `'`
		or	`"` *digit* [*digit* ...] `"O`
		or	`'` *digit* [*digit* ...] `'O`
CRI	*hexadecimal_typeless_constant*	**is**	`X'` *hex_digit* [*hex_digit* ...] `'`
		or	`X"` *hex_digit* [*hex_digit* ...] `"`
		or	`'` *hex_digit* [*hex_digit* ...] `'X`
		or	`"` *hex_digit* [*hex_digit* ...] `"X`
		or	`Z'` *hex_digit* [*hex_digit* ...] `'`
		or	`Z"` *hex_digit* [*hex_digit* ...] `"`
CRI	*binary_typeless_constant*	**is**	`B'` *digit* [*digit* ...] `'`
		or	`B"` *digit* [*digit* ...] `"`
R305	*constant*	**is**	*literal_constant*
		or	*named_constant*
R307	*named_constant*	**is**	*name*
R308	*int_constant*	**is**	*constant*
R309	*char_constant*	**is**	*constant*

The following notes pertain to the preceding format:

- *digit* must have one of the values 0 through 7 in *octal_typeless_constant*
- *digit* must have a value of 0 or 1 in *binary_typeless_constant*
- The `B`, `O`, `X`, and `Z` characters can be in uppercase or lowercase.

A value that does not have a name is a *literal constant*. The following are examples of literal constants:

```
1.23
400
( 0.0, 1.0 )
"ABC"
B'0110110' ! BOZ constant described in subsection 3.1
.TRUE.
```

No literal constant can be array-valued or of derived type. Subsection 4.3, page 82, describes the formats of literal constants in more detail.

A value that has a name is called a *named constant* and can be of any type, including a derived type. A named constant can also be array-valued. In the following statements, `X_AXIS` and `MY_SPOUSE` are examples of named constants:

```
REAL, DIMENSION(2), PARAMETER :: X_AXIS = (/ 0.0, 1.0 /)
TYPE(PERSON), PARAMETER :: MY_SPOUSE = PERSON( 39, 'PAT' )
```

Note that the entity on the right of the equal sign is not itself a constant but a constant expression. The forms for defining named constants are described in more detail in subsection 5.5.1, page 170.

Operators
3.2.4

Operators are used with operands in expressions to produce other values. The following are examples of intrinsic operators:

Operator	Representation
*	Multiplication of numeric values
//	Concatenation of character values
==	Comparison for equality (same as `.EQ.`)
`.OR.`	Logical disjunction
`.NOT.`	Logical negation

The *intrinsic_operator* s are defined as follows:

R310	*intrinsic_operator*	**is**	*power_op*
		or	*mult_op*
		or	*add_op*
		or	*concat_op*
		or	*rel_op*
		or	*not_op*
		or	*and_op*
		or	*or_op*
		or	*equiv_op*
R708	*power_op*	**is**	`**`
R709	*mult_op*	**is**	`*`
		or	`/`
R710	*add_op*	**is**	`+`
		or	`-`
R712	*concat_op*	**is**	`//`
R714	*rel_op*	**is**	`.EQ.`
		or	`.NE.`
		or	`.LT.`
		or	`.LE.`
		or	`.GT.`
		or	`.GE.`
		or	`==`
		or	`/=`
		or	`<`
		or	`<=`
		or	`>`
		or	`>=`
R719	*not_op*	**is**	`.NOT.`
CRI			`.N.`

Section 3

R720	*and_op*	**is**	`.AND.`
CRI			`.A.`
R721	*or_op*	**is**	`.OR.`
CRI			`.O.`
R722	*equiv_op*	**is**	`.EQV.`
		or	`.NEQV.`
CRI	exclusive_disjunct_op	**is**	`.XOR.`
CRI		**or**	`.X.`

The abbreviations `.A.`, `.O.`, `.N.`, and `.X.` are synonyms for `.AND.`, `.OR.`, `.NOT.`, and `.XOR.`, respectively. If the abbreviated operator is overloaded in an interface block as a defined operator, the abbreviated form of the intrinsic operator cannot be used in any scope in which the defined operator is accessible.

ANSI/ISO: The Fortran 90 standard does not specify the `.A.`, `.O.`, or `.N.`, abbreviations for the logical operators, nor does it specify the `.XOR.` operator or its `.X.` abbreviation.

You can define operators in addition to the intrinsic operators. User-defined operators begin with a period (`.`), followed by a sequence of up to 31 letters, and end with a period (`.`), except that the letter sequence must not be the same as any intrinsic operator defined by the Fortran 90 standard or the logical constants `.FALSE.` or `.TRUE.`

Statement labels
3.2.5

A *label* can be used to identify a statement. A label consists of 1 to 5 decimal digits, one of which must be nonzero. If a statement has a label, it is uniquely identified and the label can be used in `DO` constructs, `CALL` statements, branching statements, and I/O statements.

Leading zeros in a label are not significant so that the labels `020` and `20` are considered to be the same label. A statement label is local to a scoping unit. This means that a program unit can contain more than one definition of the same label as long as the labels are defined in different scoping units. For example, a main program and a contained internal subprogram can both define the same statement label. The cases in which duplicate labels can be used in the same program unit are explained as part of the general treatment of the scope of entities in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

The following are examples of statements with labels:

```
100 CONTINUE
 21 X = X + 1.2
101 FORMAT (1X, 2F10.2)
```

The Fortran 90 syntax does not permit a statement with no content, sometimes referred to as a blank statement. Such a statement is always treated as a comment; therefore, if such a statement is created, it must not be labeled. For example, each of the following lines is nonstandard Fortran 90 and generates an error message from the CF90 compiler:

```
10
         X=0;101;
```


Source form

3.3

A Fortran program consists of statements, comments, and `INCLUDE` lines; this collection of statements, comments, and `INCLUDE` lines is called *source text*. A statement consists of one or more lines of source text and is constructed from low-level syntax.

The lines within a program unit (except comment lines) and the order of the lines are, in general, significant.

Because all program units terminate with their own `END` statement, lines following such an `END` statement are never part of the preceding program unit; they are part of the program unit that follows.

There are two source forms for writing source text: free source form and fixed source form. The CF90 compiler allows you to use the `FIXED` and `FREE` compiler directives to switch from one source form to the other within a program unit. See the *CF90 Commands and Directives Reference Manual*, publication SR–3901, for information on the `FIXED` and `FREE` compiler directives.

The CF90 compiler `f90`(1) command line and compiler directives allow you to indicate which source form is being used; the default source form is fixed. Subsection 3.4, page 68, describes a way to write Fortran statements so that the source text is acceptable to both free and fixed source forms.

ANSI/ISO: The Fortran 90 standard does not describe compiler directives or command lines.

Characters that form the value of a character literal constant, a Hollerith constant, or a character string edit descriptor (quotation mark, apostrophe, or `H` edit descriptor) are in a *character context.* The characters in a character context do not include the delimiters used to indicate the beginning and end of the character constant or string. Also, the ampersands in free source form, used to indicate that a character string is being continued and used to indicate the beginning of the character string on the continued line, are never part of the character string value and thus are not in character context.

ANSI/ISO: The Fortran 90 standard does not describe the use of Hollerith constants.

The rules that apply to characters in a character context are different from the rules that apply to characters in other contexts. For example, blanks are always significant in a character context but are never significant in other parts of a program written using fixed source form.

```
CHAR = CHAR1 // "Mary K. Williams"
! The blanks within the character string
! (within the quotation marks) are significant.
! The next two statements are equivalent
! in fixed source form.
DO2I=1,N
DO 2 I = 1, N
```

Comments can contain any printable character.

Free source form
3.3.1

In free source form, there are no restrictions limiting statements to specific positions on a Fortran line (other than that the line cannot contain more than 132 characters). The blank character is significant and may be required to separate lexical tokens.

Blank characters are significant everywhere except that a sequence of blank characters outside a character context is treated as a single blank character. They can be used freely between tokens and delimiters to improve the readability of the source text. For example, the following two statements are interpreted identically:

```
SUM=SUM+A(I)
SUM = SUM + A (I)
```

Each line can contain from 0 through 132 characters.

The exclamation mark (`!`), not in character context, indicates the beginning of a comment that ends with the end of the line. A line can consist of nothing but a comment. Comments, including the `!`, are ignored and do not alter the interpretation of statements in any way.

Comment lines that begin with `!DIR$` are not treated as comments by the CF90 compiler. For more information on compiler directives, see the *CF90 Commands and Directives Reference Manual*, publication SR–3901.

A line whose first nonblank character is an exclamation mark is called a *comment line.*

The following is an example of a comment line:

```
! Begin the next iteration.
```

The following is an example of a statement with a trailing comment:

```
ITER = ITER + 1    ! Begin the next iteration.
```

An ampersand, not in a character context, is a continuation symbol and must be followed by one of the following:

- Zero or more blanks
- A comment and the end of the line. If the line following this line is not a comment line; it is a *continuation line.*

The following is an example of a continued line and a continuation line:

```
FORCE = G * MASS1 *  & ! This is a continued line.
       MASS2 / R**2 ! This is a continuation line.
```

Section 3

The CF90 compiler allows a statement to be continued with up to 99 continuation lines.

> **ANSI/ISO:** If you are using free source form, the Fortran 90 standard allows no more than 39 continuation lines within a statement.

No line can contain an ampersand as the only nonblank character before an exclamation mark. Comment lines cannot be continued; that is, the ampersand as the last character in a comment is part of the comment and does not indicate continuation.

A line with only blank characters or with no characters is treated as a comment line.

More than one statement or partial statement can appear on a line. The statement separator is the semicolon (`;`), provided it is not in a character context; multiple successive semicolons on a line with or without blanks intervening are considered as a single separator. The end of a line is also a statement separator, but a semicolon at the end of a line that is not part of a comment is considered as a single separator. Essentially, a null statement is a legal Fortran statement.

```
! The semicolon is a statement separator.
X = 1.0;  Y = 2.0

! However, the semicolon below, at the end of a
! line, is not treated as a separator and is
! ignored.
Z = 3.0;

! Also, consecutive semicolons are treated as one
! semicolon, even if blanks intervene.
Z = 3.0; ; W = 4.0
```

A label can appear before a statement, provided that it is not part of another statement, but it must be separated from the statement by at least one blank.

```
10 FORMAT(10X,2I5)                  ! 10 is a label
   IF (X == 0.0) 200 Y = SQRT(X)   ! Label 200 is
                                    ! not allowed.
```

Section 3

Any printable character can be used in character literal constants and character string edit descriptors.

The CF90 compiler supports only the ASCII character set. The *CF90 Fortran Language Reference Manual, Volume 3,* publication SR–3905, describes the ASCII character set.

3.3.1.1 The ampersand as a continuation symbol

The ampersand (&) is used as the continuation symbol in free source form. If it is the last nonblank character in a line after any comments are deleted and it is not in a character context, the statement is continued on the next line that does not begin with a comment. If the first nonblank character on the continuing line is an ampersand, the statement continues after the ampersand; otherwise, the statement continues with the first position of the line. The ampersand or ampersands used as the continuation symbols are not considered part of the statement. For example, the following statement takes two lines (one continuation line) because it is too long to fit on one line:

```
STOKES_LAW_VELOCITY = 2 * GRAVITY * RADIUS ** 2 *   &
       (DENSITY_1 - DENSITY_2) / (9 * COEFF_OF_VISCOSITY)
```

The leading blanks on the continued line are included in the statement and are allowed in this case because they are between lexical tokens.

The double ampersand convention must be used to continue a name, a character constant, or a lexical token consisting of more than 1 character split across lines. The following statement is the same statement as in the previous example:

```
STOKES_LAW_VELOCITY = 2 * GRAVITY * RADIUS ** 2 * (DEN&
       &SITY_1 - DENSITY_2) / (9 * COEFF_OF_VISCOSITY)
```

However , splitting names across lines makes the code difficult to read and is not recommended.

Ampersands can be included in a character constant. Only the last ampersand on the line is the continuation symbol, as illustrated in the following example:

```
     LAWYERS = "Jones & Clay & &
     &Davis"
```

The value of this constant is `Jones & Clay & Davis`. The first two ampersands are in a character context; they are part of the value of the character string.

3.3.1.2 Blanks as separators

Blanks in free source form cannot appear within tokens, such as names or symbols consisting of more than 1 character, but blanks can be used freely in format specifications. For example, blanks cannot appear between the characters of multicharacter operators such as `**` and `.NE.`. Format specifications are an exception because blanks can appear within edit descriptors such as `BN`, `SS`, or `TR`.

The CF90 compiler treats tabs and single blanks as equivalent in free source form, except in character literal strings.

> **ANSI/ISO:** The Fortran 90 standard does not allow tabs in Fortran source files.

A blank must be used to separate a statement keyword, name, constant, or label from an adjacent name, constant, or label. For example, the blanks in the following statements are required:

```
INTEGER SIZE
PRINT 10,N
DO I=1,N
```

Adjacent keywords require a blank separator in some cases (for example, `CASE DEFAULT`). In other cases, two adjacent keywords can be written either with or without intervening blanks (for example, `BLOCK DATA`).

Blank separators are optional in the following keywords:

- `BLOCK DATA`
- `DOUBLE PRECISION`
- `ELSE IF`
- `END BLOCK DATA`
- `END DO`
- `END FILE`
- `END IF`

Section 3

- `END INTERFACE`
- `END MODULE`
- `END PROGRAM`
- `END SELECT`
- `END SUBROUTINE`
- `END TYPE`
- `END WHERE`
- `GO TO`
- `IN OUT`
- `SELECT CASE`

Blank separators are mandatory in the following keywords:

- `CASE DEFAULT`
- `DO WHILE`
- `IMPLICIT` *type_spec*
- `IMPLICIT NONE`
- `INTERFACE ASSIGNMENT`
- `INTERFACE OPERATOR`
- `MODULE PROCEDURE`
- `RECURSIVE FUNCTION`
- `RECURSIVE SUBROUTINE`
- `RECURSIVE` *type_spec*
- *type_spec* `FUNCTION`
- *type_spec* `RECURSIVE`

Even though blanks are not mandatory in all cases, blank separators between statement keywords make the source text more readable and clarify the statements. Generally, if common rules of English text are followed, everything will be correct. For example, blank separators in the following statement make them quite readable, even though the blank between the keywords `RECURSIVE` and `FUNCTION` in the first statement is the only one that is required:

```
RECURSIVE FUNCTION F(X)
DOUBLE PRECISION X
END FUNCTION F
```

Sample program, free source form
3.3.1.3

The following is a sample program in free source form. Note that the numbers and the line at the top are not part of the program; the vertical bars to the left of the program are also not part of the program. These graphics are included to show the columns this program uses in free source form.

```
 123456789.......
-----------------------------------------------------
|PROGRAM LEFT_RIGHT
| REAL  X(5), Y(5)
|       ! Print arrays X and Y
|       PRINT 100, X, Y
|       100 FORMAT (F10.1, F10.2, F10.3, F10.4,  &
|                   F10.5)
|   . . .
|END
```

Section 3

Fixed source form
3.3.2

Fixed source form is position oriented on a line using the historical Fortran conventions for position that were used on punched cards.

By default, statements or parts of statements must be written between positions 7 and 72, inclusively. You can use the `-N` option on the `f90`(1) command line to extend the line length, as follows:

- If `-N 80` is specified, statements can occur in columns 7 through 80.
- If `-N 132` is specified, statements can occur in columns 7 through 132.

Regardless of the `-N` specification, character positions 1 through 6 are reserved for special purposes.

ANSI/ISO: The Fortran 90 standard does not allow a compiler to recognize characters beyond column 72.

Blanks are not significant in fixed source form except in a character context. For example, the following two statements are identical:

```
D O  10  I = 1, L O O P E N D
DO 10 I = 1, LOOPEND
```

A `C` or `*` in position 1 identifies a comment. In this case, the entire line is a comment and is called a *comment line*. A `!` in any position except position 6 and not in a character context indicates that a comment follows to the end of the line. Comments are not significant.

A line with only blank characters or with no characters is treated as a comment line.

Multiple statements on a line are separated by one or more semicolons. Semicolons can occur at the end of a line, and these are ignored.

Any character (including `!` and `;`) other than blank or 0 in position 6 indicates that the line is a continuation of the previous line. Such a line is called a *continuation line*. The text on the continuation line begins in position 7. There can be no more than 99 continuation lines for one statement in fixed source form. The first line of a continued statement is called the *initial line*.

ANSI/ISO: If you are using fixed source form, the Fortran 90 standard allows no more than 19 continuation lines within a statement.

Statement labels can appear only in positions 1 through 5. A label can appear only on the initial line of a continued statement. Thus, positions 1 through 5 of continuation lines must contain blanks.

The CF90 compiler allows you to continue an `END` statement, as follows:

```
 E
&N
&D
```

ANSI/ISO: The Fortran 90 standard states that an `END` statement must not be continued.

Section 3

The characters `END` can appear as the only characters on the initial line of a statement, as follows:

```
 END
&FILE(10)
```

> **ANSI/ISO:** The Fortran 90 standard states that `END` cannot be an initial line of a statement other than an `END` statement.

Any character from the CF90 compiler character set (including graphic and control characters) can be used in character literal constants and character edit descriptors. Although the Fortran 90 standard permits a processor to limit the use of control characters (such as the newline) to such character contexts, the CF90 compiler imposes no such limitation.

Tab character
3.3.2.1

You can substitute the tab character for spaces at the beginning of a line, but it is not actually converted to spaces. The compiler uses the tab as follows:

- If a tab is the first character on a line, the next character determines how the line is interpreted. A nonzero digit indicates a continuation line; otherwise, this line is the initial line of a statement.
- A statement label, if present, must precede the first tab.
- The compiler treats the tab character in a statement the same way it treats a blank character.

A tab character is not converted to spaces, so the exact visual placement of tabbed statements depends on the utility you use to edit or display text.

> **Note:** A tab counts as 1 character even though it may expand to more than one space in the listing or editor. Thus, statements that include tabs may appear to have data beyond column 72 (or 80) in the editor or listing.

Sample program, fixed source form
3.3.2.2

The following is a sample program in fixed source form. Note that the numbers and the line at the top are not part of the program; the vertical bars to the left of the program are also not part of the program. These graphics are included to show the columns this program uses in fixed source form.

```
 123456789.....
---------------------------------------------
|      PROGRAM LEFT_RIGHT
|      REAL  X(5), Y(5)
|C     Print arrays X and Y
|      PRINT 100, X, Y
|  100 FORMAT (F10.1, F10.2, F10.3, F10.4,
|     1        F10.5)
|         . . .
|      END
```

Section 3

Portable source form
3.4

For portability in many cases, such as an included file, it is desirable to use a form of the source code that is valid and equivalent for either free source form or fixed source form. Such a source form can be written by obeying the following rules and restrictions:

- Limit labels to positions 1 through 5, and statements to positions 7 through 72. These are the limits required in fixed source form.
- Treat blanks as significant. Because blanks are ignored in fixed source form, using the rules of free source form will not impact the requirements of fixed source form.
- Use the exclamation mark (`!`) for a comment, but do not place it in position 6, which indicates continuation in fixed source form. Do not use the `C` or `*` forms for a comment.
- To continue statements, use the ampersand in both position 73 of the line to be continued and in position 6 of the continuation. Positions 74 to 80 must remain blank or contain only a comment. Positions 1 through 5 must be blank. The first ampersand continues the line after position 72

in free source form and is ignored in fixed source form. The second ampersand indicates continuation in fixed source form and in free source form indicates that the text for the continuation of the previous line begins after the ampersand.

- The CF90 compiler allows you to switch between fixed and free source forms within a file or include file by using the `FIXED` and `FREE` compiler directives. Begin compiler directives with the characters `!DIR$` in column 1. For more information on compiler directives, see the *CF90 Commands and Directives Reference Manual*, publication SR–3901.

Sample program, use with either source form

3.4.1

The following is a sample program that is acceptable in either fixed or free source form. Note that the numbers and the line at the top are not part of the program; the vertical bars to the left of the program are also not part of the program. These graphics are included to show the columns this program uses in fixed source form.

```
123456789.....                                        73
-------------------------------------------.....----
|      PROGRAM LEFT_RIGHT
|      REAL  X(5), Y(5)
|!     Print arrays X and Y
|      PRINT 100, X, Y
|  100 FORMAT (F10.1, F10.2, F10.3, F10.4,            &
|     &        F10.5)
|          . . .
|      END
```

The `INCLUDE` line

3.5

Source text can be imported from another file and included within a program file during processing. An `INCLUDE` line consists of the keyword `INCLUDE` followed by a character literal constant.

```
INCLUDE  'MY_COMMON_BLOCKS'
```

Section 3

The specified text is substituted for the `INCLUDE` line during compilation and is treated as if it were part of the original program source text. The CF90 compiler allows you to specify search path names on the `f90`(1) command line for locating files to be included. See the *CF90 Commands and Directives Reference Manual*. publication SR–3901, for more information.

The `INCLUDE` line provides a convenient way to include source text that is the same in several program units. For example, the specification of interface blocks or objects in common blocks may constitute a file that is referenced in the `INCLUDE` line.

The format for an `INCLUDE` line is as follows:

> `INCLUDE` *character_literal_constant*

The character literal constant used must not have a kind parameter that is a named constant.

The `INCLUDE` line is a directive (but not a Cray Research compiler directive) to the compiler; it is not a Fortran statement.

The `INCLUDE` line is placed where the included text is to appear in the program.

The `INCLUDE` line must appear on one line with no other text except possibly a trailing comment. There must be no statement label.

The `INCLUDE` lines can be nested. That is, a second `INCLUDE` line may appear within the text to be included. The Fortran 90 standard does not specify the permitted level of nesting, and the CF90 compiler imposes no limit. The text inclusion must not be recursive at any level; for example, included text `A` must not include text `B`, which includes text `A`.

A file intended to be referenced in an `INCLUDE` line must not begin or end with an incomplete statement.

An example of a program unit with an `INCLUDE` line follows:

```
PROGRAM MATH
REAL, DIMENSION(10,5,79) :: X, ZT
!  Some arithmetic
INCLUDE 'FOURIER'
!  More arithmetic
   . . .
END
```

The source text in the file `FOURIER` in effect replaces the `INCLUDE` line.

Low-level syntax

3.6

The basic lexical elements of the language consist of the classes *character*, *name*, *constant*, *intrinsic_operator*, *defined_operator*, and *label*; these are defined by rules R301 through R313, some of which were defined in previous subsections. R311 and subsequent rules are defined as follows:

R311	*defined_operator*	**is**	*defined_unary_op*
		or	*defined_binary_op*
		or	*extended_intrinsic_op*
R704	*defined_unary_op*	**is**	`.`*letter* [*letter*] ... `.`
R724	*defined_binary_op*	**is**	`.`*letter* [*letter*] ... `.`
R312	*extended_intrinsic_op*	**is**	*intrinsic_operator*
R313	*label*	**is**	*digit* [*digit* [*digit* [*digit* [*digit*]]]]

Section 3

Data Types [4]

Fortran was designed to give scientists and engineers an easy way to solve problems by using computers. Statements could be presented that looked like formulas or English sentences. For example, the following statement might be performing typical numeric calculations:

```
X = B + A * C
```

As another example, the following statement could specify that a certain action is to be taken based on a logical decision:

```
IF (LIMIT_RESULTS .AND. X .GT. XMAX) X = XMAX
```

And the following statement could be used to communicate the results of a calculation to a scientist or engineer in a meaningful way:

```
PRINT *, "CONVERGENCE REACHED"
```

Each of these statements performs a task that uses a different type of data.

Task	Data Type
Calculating typical numeric results	Numeric data
Making decisions	Logical data
Explaining	Character data

These are the commonly needed data types, and the Fortran standard provides for them.

The CF90 compiler supports additional data types, and this preserves compatibility with the Cray Research CF77 compiling system and with other vendor systems. These additional types are as follows:

- Cray pointer
- Cray character pointer
- Boolean (or typeless)

ANSI/ISO: The Fortran 90 standard does not specify Cray pointer, Cray character pointer, or Boolean data types.

Section 4

Anything provided by the language is *intrinsic* to the language. Types that are not intrinsic to the language can be specified by a programmer; this facility is new in Fortran 90. The new types are built of (or derived from) the intrinsic types and thus are called *derived types.* The Fortran 90 data types are categorized in Figure 4.

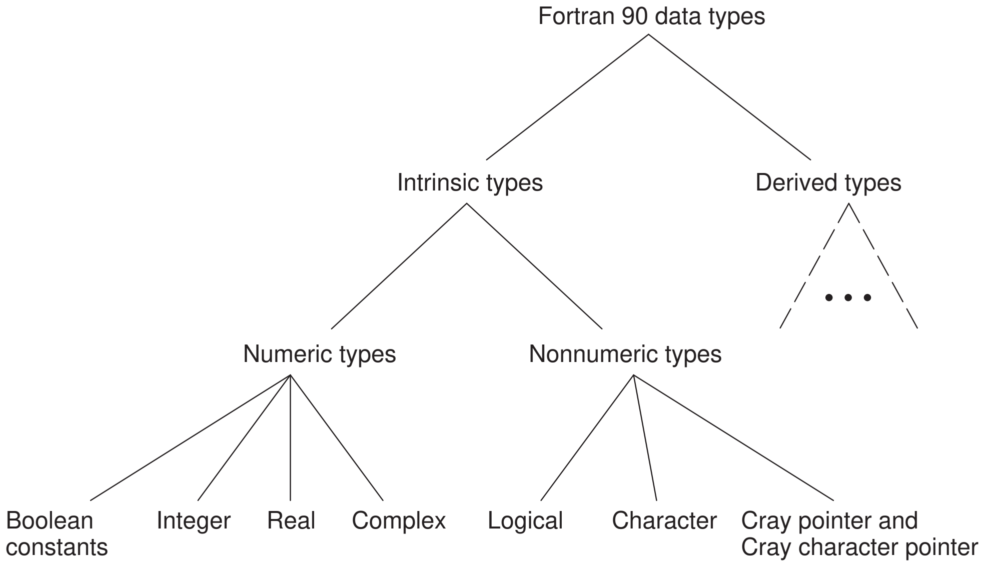

Figure 4. Fortran 90 data types

As the following list shows, the type of the data determines the operations that can be performed on it:

Operations	Data Type
Addition, subtraction, multiplication, division, exponentiation, negation, comparison, masking expressions	Real, complex, integer, Boolean
Negation, conjunction, disjunction, and equivalence	Logical
Concatenation, comparison	Character

Operations	Data Type
User defined	User defined
Addition, subtraction, and `LOC( )` function	Cray pointer, Cray character pointer

The intrinsic types have the appropriate built-in (intrinsic) operations. You must define the operations performed on user-defined data types.

This section explains the Fortran 90 data types. It describes each of the intrinsic types, and it explains derived types and the facilities provided by the language that allow you to define types and declare and manipulate objects of these types in ways that are analogous to the ways in which objects of the intrinsic types can be manipulated.

Building the data environment for a problem solution

4.1

When envisioning a computer solution to a problem you focus on the operations that must be performed and the order in which they must be performed. It is a good idea, however, to consider the variables you will need before you determine all the computational steps that are required. The variables that are chosen, together with their types and attributes, sometimes determine the course of computation, particularly when variables of user-defined type are involved.

Choosing the type and other attributes of a variable

4.1.1

There are several decisions to make about a variable in a program. If the variable is of an intrinsic type, the intended use of the variable will readily determine its type, making this is an easy decision. While type is the most important attribute of a variable, there are other attributes. Certainly it is necessary to decide very early whether the variable is to be a single data object (a scalar) or an array. Fortran 90 provides many new facilities for manipulating arrays as objects, making it possible to specify computations as straightforward array operations that, in FORTRAN 77, required program loops.

Because Fortran 90 provides allocatable arrays and pointers, it is no longer necessary to decide at the outset how big an array must be. In fact, determining sizes can be postponed until the finished program is executed, when sizes can be read in as input or calculated. Setting aside space for an array can be deferred until the appropriate size needed for a particular calculation is known.

Another decision that can be made about a variable is its accessibility. Control of accessibility is a new feature available in modules. If the variable is needed only within the module, then it can be kept private or hidden from other program units which prevents it from being corrupted inadvertently . This new feature can be used to make Fortran 90 programs safer and more reliable.

In addition to type, dimensionality, dynamic determination, and accessibility , there are other attributes that can be applied to data objects. The attributes that are permitted depend on where and how the object is to be used; for example, there are a number of attributes that can be applied only to subprogram arguments. Section 5, which begins on page 131, describes all of the attributes of data objects.

Choosing the kind of a variable of intrinsic type

4.1.2

After the type of a variable is decided, you may need to consider which kind of the type to use. *Kind* is a technical term that is new with Fortran 90. Each of the intrinsic types can be specified with a *kind parameter* that selects a processor-dependent representation of objects of that type and kind. If no kind parameter is specified, the default kind is assumed. FORTRAN 77 did not have kind parameters but it provided two kinds for the real type: `REAL` and `DOUBLE PRECISION`. It treated double-precision real as a separate type. Fortran 90, while remaining compatible with FORTRAN 77, treats it as a separate kind of real. There are two ways to declare real variables in Fortran 90: one is with a `REAL` statement specifying a nondefault kind, and the other is with a `DOUBLE PRECISION` statement. FORTRAN 77 provided only one kind for the other four types (integer , complex, logical, and character).

> **Note:** Depending on your hardware platform, the CF90 compiler may support more than one kind for each data type.

Section 4

Fortran 90 requires a processor to support at least two kinds for the real and complex types and at least one kind for the other three intrinsic types. The CF90 compiler supports several kinds of real, complex, logical, and integer data types.

The Fortran 90 kind parameters for each of the intrinsic types addresses several problems that existed with FORTRAN 77.

- Real. Programs with `REAL` and `DOUBLE PRECISION` declarations are not numerically portable across machine architectures with different word sizes. The CF90 compiler chooses a representation for the real type that is efficient on the target machine. For example, a representation that fits into 32 bits is used on machines with 32-bit words, and a representation that fits into 64 bits is used on machines with 64-bit words.

  In earlier versions of Fortran, if the target machine was a 32-bit machine and 64 bits were required for the numerical stability of the algorithm, a user typically would have specified `DOUBLE PRECISION` declarations. If the program were subsequently moved to a 64-bit machine, the `DOUBLE PRECISION` declarations would have to be changed to `REAL` declarations. This is because a 128-bit representation was not needed and would have degraded program performance.

  However , with Fortran 90, a kind parameter gives you access to and control over the use of different machine representations of real values in order to make a program more portable. For example, a kind parameter in a `REAL` declaration can specify a required minimum precision, as follows:

  ```
  REAL(KIND=SELECTED_REAL_KIND(10,50)) :: REAL_VALUE
  ```

  When a program is run on a 32-bit machine, it uses two words to contain variable `REAL_VALUE`. When the same program (without any changes) is run on a 64-bit machine, one word is used to contain variable `REAL_VALUE`.

- Complex . The FORTRAN 77 standard did not require a double-precision complex type. This is an inconsistency that is corrected by the kind parameters specified in the Fortran 90 standard. Fortran 90 does not need a `DOUBLE PRECISION COMPLEX` specifier because it uses a `COMPLEX`

Section 4

attribute with a nondefault kind parameter to specify double-precision complex. As with the real type, more than two representations for complex may be available on a given platform, but at least two are provided.

- Character . The FORTRAN 77 character data type had an underlying machine representation of a single byte per character. This is adequate to represent 2^8 or 256 different characters, which is enough for the ASCII character set. With kind type parameter values, a single character could occupy more than one byte.

  The CF90 compiler, however, supports only the ASCII character set, so it has no nondefault character kind.

- Integer . One motivation for alternative representations of integer data is to provide an integer kind with a very large range. As with the logical data type, only one representation is required in a standard-conforming Fortran 90 processor, but more are permitted. The CF90 compiler supports several integer kinds.

Choosing to define a type for a variable
4.1.3

Sometimes it is easier to think about an essential element of a problem as several pieces of related data, not necessarily all of the same type. Arrays can be used to collect homogeneous data (all of the same type) into a single variable. In Fortran 90, a *structure* is a collection of nonhomogeneous data in a single variable. To declare a structure, it is first necessary to define a type that has components of the desired types. The structure is then declared as an object of this user-defined (or derived) type. An example of objects declared to be of user-defined type was given in subsection 2.3.1, page 22. It is repeated here. First a type, named PATIENT, is defined; then two structures JOHN_JONES and SALLY_SMITH are declared.

```
TYPE PATIENT
   INTEGER          PULSE_RATE
   REAL             TEMPERATURE
   CHARACTER *300  PROGNOSIS
END TYPE PATIENT

TYPE(PATIENT)      JOHN_JONES, SALLY_SMITH
```

Type PATIENT has three components, each of a different intrinsic type (integer , real, and character). In practice, a type of this nature probably would have even more components, such as the patient's name and address, insurance company, room number in the hospital, and so on. For purposes of illustration, three components are sufficient. JOHN_JONES and SALLY_SMITH are structures (or variables) of type PATIENT. A type definition indicates names, types, and attributes for its components; it does not declare any variables that have these components. Just as with the intrinsic types, a type declaration is needed to declare variables of this type. Because there is a type definition, though, any number of structures can be created that have the components specified in the type definition for PATIENT; subprogram arguments and function results can be of type PATIENT; there can be arrays of type PATIENT; and operations can be defined that manipulate objects of type PATIENT. Thus, the derived-type definition can be used merely as a way to specify a pattern for a particular collection of related but nonhomogeneous data; but, because the pattern is specified by a type definition, a number of other capabilities are available.

What is meant by *type* in Fortran?

4.2

Knowing exactly what is meant by *type* in Fortran becomes more important now that you can define types in addition to the intrinsic types. A data type provides a means to categorize data and determine which operations can be applied to the data to get desired results. The following exist for each data type:

- A name
- A set of values
- A set of operations
- A form for constants of the intrinsic types and constructors for derived types

Section 4

Data type names

4.2.1

Each of the intrinsic types has a name supplied by the standard. The names of derived types must be supplied in type definitions. The name of the type is used to declare entities of the type unless the programmer chooses to let the processor determine the type of an entity implicitly by the first character of its name. Section 5, page 131, describes declarations and implicit typing.

Data type values

4.2.2

Each type has a set of valid values. The logical type has only two values: true and false. The integer type has a set of integral numeric values that can be positive, negative, or zero. For complex or derived types, the set of valid values is the set of all combinations of the values of the individual components.

The kind of an intrinsic type determines the set of valid values for that type and kind. For example, there is more than one integer data type: the default type and shorter integer types. The shorter integer types have values that are a subset of the default integer values. The kind of a type is referred to as a *kind parameter* or *kind type parameter* of the type. The character data type has a length parameter as well as a kind parameter. The length parameter specifies the number of characters in an object, and this determines the valid values for a particular character object. Derived types do not have parameters, even though their components may.

Data type operations

4.2.3

For each of the intrinsic data types, a set of operations with corresponding operators is provided by the language. These are described in section 7, which begins on page 233.

You can specify new operators and define operations for the new operators. The form of a new operator is an alphabetic name of your choice delimited by periods. These new operators are analogous to intrinsic operators such as `.GT.`, `.AND.`, and `.NEQV.`. For example, you might specify and define the operations `.PLUS.`, `.REMAINDER.`, and `.REVERSE.`. In defining the

Section 4

operation, the types of allowable operands must be specified. Such new operations can apply to objects of intrinsic type and in these cases extend the set of operations for the type. You would more frequently be defining operations for objects of derived type.

You cannot redefine an intrinsic operation, but you can define meanings for intrinsic operator symbols when at least one operand is not of an intrinsic type or for intrinsic operands for which the intrinsic operation does not apply. For example, consider the expression `A + B`. If both `A` and `B` are of numeric type, the operation is defined intrinsically and cannot be redefined. However, if either `A` or `B` is of derived type or nonnumeric type, the plus operation between `A` and `B` is not defined intrinsically, and you can provide a meaning for the operation. New operations are defined by functions with the `OPERATOR` interface. These are described in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

Assignment is defined intrinsically for each intrinsic and derived type. Structure assignment is component-by-component intrinsic or pointer assignment, though this can be replaced by a defined assignment. No other intrinsically defined assignment, including array assignment, can be redefined. Beyond this, any assignment between objects of different type may be defined with the `ASSIGNMENT` interface as described in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

Forms for constants and constructors
4.2.4

The language specifies the syntactic forms for literal constants of each of the intrinsic types. Syntactic mechanisms (called derived-type constructors) specify derived-type values. As shown in Figure 5, the form indicates both the type and a particular member of the set of valid values for the type.

Syntax	Type	Value
1	integer	1
103.1 or 1.031E2	real	103.1
(1.0, 1.0)	complex	1 + *i*
.TRUE.	logical	true
“Hello”	character	Hello
PATIENT (70, 99.7, “Recovering”)	patient	70 99.7 Recovering

Figure 5. Forms of constants and constructors

If a constant is not of default kind, some indication of its kind must be included in its syntactic form. This form is the default literal constant separated from the kind value by an underscore. Kind specifications follow integer , real, and logical values. Kinds are known to the CF90 compiler as integer values, but if a program is to be portable, the actual numbers should not be used because the kind values are processor dependent. Instead, a kind value should be assigned to a named constant, and this name should be used.

Real	`1.3141592653589_DOUBLE`
Complex	`(1.75963_HIGH, -2.0)`

`DOUBLE` and `HIGH` are named constants for kind values.

The kind of a complex constant is determined by the kind of its parts. Subsection 4.3.3.3, page 95, describes the form for complex literal constants.

Intrinsic data types

4.3

The FORTRAN 77 data types were based on a storage model in which a real, integer, and logical object each were represented in a numeric storage unit, and a complex and double-precision object each were represented in two numeric storage units. A character in FORTRAN 77 was represented in a character storage unit, which is different from a numeric storage unit. Fortran 90, while remaining compatible with FORTRAN 77, extends this underlying model. The default kinds for Fortran 90 intrinsic types conform to the FORTRAN 77 model. That is, the storage units for default real kind, default integer kind, and default logical kind must all be the same. Default complex (which is really two default reals) requires two of these storage units, and double-precision real requires two of these storage units.

> **Note:** This subsection describes each of the intrinsic types. The descriptions include a simple statement form to show how objects of these types can be declared. These simple forms are not complete; if they are used to construct statements, the statements will be correct, but other variations are permitted. Subsection 5.1, page 133, contains the complete formats. The kind parameter that appears in the formats is limited to a scalar integer initialization expression, which is described in subsection 7.2.9.2, page 282.

Integer type
4.3.1

The name of the integer type is `INTEGER`. The following format shows how integer objects can be declared:

```
INTEGER [ ( [ KIND = ] kind_param ) ] [ , attribute_list :: ] entity_list
```

Examples:

```
INTEGER X
INTEGER COUNT, K, TEMPORARY_COUNT
INTEGER(SHORT) PARTS
INTEGER, DIMENSION(0:9) :: SELECTORS, IX
```

Values
4.3.1.1

The integer data type has values that represent a subset of the mathematical integers. The intrinsic inquiry function `RANGE` provides the decimal exponent range for integers of the kind of its argument. Only one kind of integer is required by the standard, but the CF90 compiler supports several. Values that overflow in storage may be truncated.

If *kind_param* is specified, it must have one of the following values: 1, 2, 4, 6, or 8. The default *kind_param* is specific to your hardware platform. These values are shown in Table 2, page 84. See the *CF90 Commands and Directives Reference Manual*, publication SR–3901, for information on changing default values.

Warning: Specifying `KIND=6` is obsolescent. When specified, a warning message is generated. On SPARC systems, when `KIND=6` is specified, `KIND=4` is used. The `KIND=6` kind type parameter will be removed in the CF90 3.0 release.

Table 2. Integer kind values

kind_param	Value range	Size / Storage	Platforms
1	$-2^7 < n < 2^7$	8 bits / 64 bits	PVP
1	$-2^7 < n < 2^7$	8 bits / 32 bits	CRAY T3E, SPARC
2	$-2^{15} < n < 2^{15}$	16 bits / 64 bits	PVP
2	$-2^{15} < n < 2^{15}$	16 bits / 32 bits	CRAY T3E, SPARC
4	$-2^{31} < n < 2^{31}$	32 bits / 64 bits	PVP
4	$-2^{31} < n < 2^{31}$	32 bits / 32 bits	CRAY T3E
4 (default)	$-2^{31} < n < 2^{31}$	32 bits / 32 bits	SPARC
6 (default)	$-2^{45} < n < 2^{45}$	46 bits / 64 bits	PVP (not IEEE) (obsolescent)
6 (default)	$-2^{50} < n < 2^{50}$	50 bits / 64 bits	CRAY T90 (IEEE) (obsolescent)
8	$-2^{52} < n < 2^{52}$	53 bits / 64 bits	PVP (with `-O fastint` specified)
8	$-2^{63} < n < 2^{63}$	64 bits / 64 bits	PVP
8 (default)	$-2^{63} < n < 2^{63}$	64 bits / 64 bits	CRAY T3E
8	$-2^{63} < n < 2^{63}$	64 bits / 64 bits	SPARC

In Table 2, the *Size* information refers to the size according to the integer model defined on the `MODELS`(3) man page.

Section 4

Table 3 shows power-of-10 values that approximate the power-of-2 values shown in Table 2:

Table 3. Exponent equivalents

2^n	10^k
2^7	10^2
2^{15}	10^4
2^{31}	10^9
2^{45}	10^{13}
2^{52}	10^{15}
2^{63}	10^{18}

The `RANGE` intrinsic function returns the decimal exponent range of a given number . For more information on this intrinsic function, see the `RANGE`(3) man page.

The `KIND` intrinsic function can be used to determine the kind parameter of its integer argument.

The `SELECTED_INT_KIND` intrinsic function returns the integer kind parameter required to represent as many decimal digits as are specified by the function argument. If there is no such integer type available on your system, –1 is returned. The following example declares `I` and `J` to be integer objects with a representation method that permits at least five decimal digits; that is, it includes all integers between -10^5 and 10^5:

```
INTEGER (SELECTED_INT_KIND (5)) I, J
```

4.3.1.2 Operators

There are both binary and unary intrinsic operators for the integer type. Binary operators have two operands and unary operators have only one. The binary arithmetic operations for the integer type are: `+`, `−`, `*`, `/`, and `**`. The unary arithmetic operations are `+` and `−`. The relational operations (all binary) are: `.LT.`, <, `.LE.`, <=, `.EQ.`, ==, `.NE.`, /=, `.GE.`, >=, `.GT.`, and >. The result of an intrinsic arithmetic operation on integer operands is an integer entity; the result of an intrinsic relational operation is a logical entity of default logical kind.

Format for constant values
4.3.1.3

An integer constant is a string of decimal digits, optionally preceded by a sign, and optionally followed by an underscore and a kind parameter. An integer constant that must not have a kind type parameter is defined as follows:

R401	*signed_digit_string*	**is**	[*sign*] *digit_string*
R402	*digit_string*	**is**	*digit* [*digit*] ...

The format of a signed integer literal constant (which may have a kind type parameter) is defined as follows:

R403	*signed_int_literal_constant*	**is**	[*sign*] *int_literal_constant*
R404	*int_literal_constant*	**is**	*digit_string* [_ *kind_param*]
R405	*kind_param*	**is**	*digit_string*
		or	*scalar_int_constant_name*
R406	*sign*	**is**	+
		or	−

The signed integer literal constant often takes the following format:

[*sign*] *digit_string* [_ *kind_param*]

In the following examples, assume that LONG and SHORT are named constants with values that are valid integer kind parameters:

```
42
9999999999999999999999_LONG
+64
10000000
-258_SHORT
```

Integer constants are interpreted as decimal values. However, in a DATA statement, it is possible to initialize an object with a value that is presented as if it had a nondecimal base. The allowed forms are unsigned binary, octal, and hexadecimal constants, and are defined as follows:

R407	*boz_literal_constant*	**is**	*binary_constant*
		or	*octal_constant*
		or	*hex_constant*

The CF90 compiler supports binary, octal, and hexadecimal constants in other contexts; for more information on these forms see subsection 4.3.6, page 100.

A *binary_constant* is defined as follows:

R408	*binary_constant*	**is**	B ' *digit* [*digit*] ... '
		or	B " *digit* [*digit*] ... "

You must specify 0 or 1 for *digit* .

An *octal_constant* is defined as follows:

R409	*octal_constant*	**is**	O ' *digit* [*digit*] ... '
		or	O " *digit* [*digit*] ... "

You must specify a value from 0 through 7 for *digit* .

A *hex_constant* is defined as follows:

R410	*hex_constant*	**is**	`Z` `'` *hex_digit* [*hex_digit*] ... `'`
		or	`Z` `"` *hex_digit* [*hex_digit*] ... `"`
R411	*hex_digit*	**is**	*digit*
		or	`A`
		or	`B`
		or	`C`
		or	`D`
		or	`E`
		or	`F`

As shown in R411, you must specify a value from 0 through 9 or one of the letters `A` through `F` (representing the decimal values 10 through 15) for *digit* . The CF90 compiler supports lowercase letters, so the hexadecimal digits `A` through `F` can be represented by their lowercase equivalents, `a` through `f`.

In these constants, the binary, octal, and hexadecimal digits are interpreted according to their respective number systems. Examples (all of which have a value equal to the decimal value 10) are as follows:

```
B"1010"
O'12'
Z"A"
```

Section 4

Real type

4.3.2

The name of the real data type is `REAL`. The name `DOUBLE PRECISION` is used for another kind of the real type. You can use one of the following formats to declare objects of real type:

`REAL [ ( [ KIND = ]` *kind_param* `) ] [ ,` *attribute_list* `:: ]` *entity_list*

```
DOUBLE PRECISION [ , attribute_list :: ] entity_list
```

Examples:

```
REAL X, Y
REAL(KIND = HIGH), SAVE :: XY(10, 10)
REAL, POINTER :: A, B, C
DOUBLE PRECISION DD, DXY, D
```

Values
4.3.2.1

The values of the real data type approximate the mathematical real numbers. The set of values varies from processor to processor. The Fortran 90 standard requires a processor to support at least two approximation methods for the real type. The CF90 compiler provides three, so there are three kind values for the real type.

If *kind_param* is specified, it must have one of the following values: 4, 8, or 16. The default *kind_param* and the values associated with the kind values are specific to your hardware platform. These values are shown in Table 4, page 90. See the *CF90 Commands and Directives Reference Manual*, publication SR–3901, for information on changing default *kind_param* values.

Section 4

Table 4. Real and complex kind values

kind_param	Value range	Size and Storage	Platforms
4	$2^{-8188} \le n < 2^{8189}$	64 bits	PVP (not IEEE)
4	$2^{-125} \le n < 2^{128}$	32 bits	CRAY T3E
4	$2^{-1021} \le n < 2^{1024}$	64 bits	CRAY T90 (IEEE)
4 (default)	$2^{-125} \le n < 2^{128}$	32 bits	SPARC
8 (default)	$2^{-8188} \le n < 2^{8189}$	64 bits	PVP (not IEEE)
8 (default)	$2^{-1021} \le n < 2^{1024}$	64 bits	CRAY T3E, CRAY T90 (IEEE)
8	$2^{-1021} \le n < 2^{1024}$	64 bits	SPARC
16	$2^{-8188} \le n < 2^{8189}$	128 bits	PVP (not IEEE)
16 †	$2^{-16381} \le n < 2^{16384}$	128 bits	CRAY T3E
16	$2^{-16381} \le n < 2^{16384}$	128 bits	CRAY T90 (IEEE), SPARC

† Deferred implementation

Section 4

Table 5 shows power-of-10 values that approximate some of the power-of-2 values shown in Table 4:

Table 5. Exponent equivalents

2^n	10^k
2^{128}	10^{38}
2^{1024}	10^{308}
2^{8189}	10^{2466}
2^{16384}	10^{4932}

The KIND intrinsic function can be used to determine the kind parameter of its real argument. The intrinsic functions PRECISION and RANGE return the decimal precision and exponent range of the approximation method used for the kind of the argument. The intrinsic function SELECTED_REAL_KIND returns the kind value required to represent as many digits of precision as specified by the first argument and the decimal range specified by the optional second argument.

The following example declares X to have at least five decimal digits of precision and no specified minimum range:

```
REAL(SELECTED_REAL_KIND(5)) X
```

The following example declares Y to have at least eight decimal digits of precision and a range that includes values between 10^{-70} and 10^{70} in magnitude:

```
REAL(SELECTED_REAL_KIND(8, 70)) Y
```

Operators
4.3.2.2

The intrinsic binary arithmetic operators for the real type are: +, −, *, /, and **. The intrinsic unary arithmetic operators are: + and −. The relational operators are: .LT., <, .LE., <=, .EQ., ==, .NE., /=, .GE., >=, .GT., and >. The result of an intrinsic arithmetic operation on real operands is a real entity. If one of the operands of an arithmetic operation is an integer entity, the result is still a real entity. The result of an intrinsic relational operation is a logical entity of default logical kind.

Forms for constants
4.3.2.3

A real constant is distinguished from an integer constant by containing either a decimal point, an exponent, or both. The format for a *signed_real_literal_constant* is defined as follows:

R412	*signed_real_literal_constant*	**is**	[*sign*] *real_literal_constant*
R413	*real_literal_constant*	**is**	*significand* [*exponent_letter exponent*] ≡ ≡ [_ *kind_param*]
		or	*digit_string exponent_letter exponent* ≡ ≡ [_ *kind_param*]
R414	*significand*	**is**	*digit_string* **.** [*digit_string*]
		or	**.** *digit_string*
R415	*exponent_letter*	**is**	`E`
		or	`D`
R416	*exponent*	**is**	*signed_digit_string*

A signed real literal constant can take one of the following forms:

[*sign*] *digit_string exponent_letter exponent* [_ *kind_param*]

[*sign*] *whole_part* **.** [*fraction_part*] [*exponent_letter exponent*] [_ *kind_param*]

[*sign*] **.** *fraction_part* [*exponent_letter exponent*] [_ *kind_param*]

For *whole_part*, specify a *digit_string* (R402). For *fraction_part*, specify a *digit_string* (R402). For *exponent*, specify a *signed_digit_string* (R401).

If both a *kind_param* and an *exponent_letter* are present, the *exponent_letter* must be `E` or `e`. If a *kind_param* is present, the real constant is of that kind; if a `D` or `d` exponent letter is present, the constant is of type double-precision real; otherwise, the constant is of type default real.

If you have disabled double-precision arithmetic by specifying `-d p` on the `f90` command, `D` is treated as `E`.

A real constant can have more decimal digits than will be used to approximate the real number.

Examples of signed real literal constants are as follows:

```
-14.78
+1.6E3
2.1
-16.E4_HIGH
0.45_LOW
.123
3E4
2.718281828459045D0
```

In the preceding example, the parameters `HIGH` and `LOW` must have been defined, and their values must be kind parameters for the real data type permitted by the CF90 compiler.

If a real literal constant has a kind parameter, it takes precedence over an exponent letter. Consider the following example:

```
1.6E4_HIGH
```

The example's code fragment will be represented by the method specified for `HIGH`, even though `1.6E4` would be represented by a different method.

Complex type

4.3.3

The name of the complex type is `COMPLEX`. A format for declaring objects of this type is as follows:

`COMPLEX [ ( [ KIND = ]` *kind_param* `) ] [ ,` *attribute_list* `:: ]` *entity_list*

Examples:

```
COMPLEX CC, DD
COMPLEX(KIND = single), POINTER :: CTEMP(:)
```

4.3.3.1 Values

The complex data type has values that approximate the mathematical complex numbers. A complex value is a pair of real values; the first is called the *real part* and the second is called the *imaginary part*. Each approximation method used to represent data entities of type real is available for entities of type complex with the same kind parameter values. Therefore, there are three approximation methods for complex.

When a complex entity is declared with a kind specification, this kind is used for both parts of the complex entity. There is no special double-precision complex declaration, as such. If no kind parameter is specified, the entity is of type default complex which corresponds to default real. The `SELECTED_REAL_KIND` intrinsic function may be used in a declaration of a complex object.

If *kind_param* is specified, it must have one of the following values: 4, 8, or 16. The default kind is specific to your hardware platform. The values, the ranges (for both the real and imaginary portions of the number), and the defaults supported are the same as those for type real, and these values are shown in Table 4, page 90. See the *CF90 Commands and Directives Reference Manual*, publication SR–3901, for information on changing default *kind_param* values.

For information on power of 10 and power of 2 equivalent values, see Table 5, page 90.

In the following example, `CX` must be represented by an approximation method with at least 8 decimal digits of precision and at least a decimal exponent range between 10^{-70} and 10^{70} in magnitude for the real and imaginary parts:

```
COMPLEX(SELECTED_REAL_KIND(8, 70)) CX
```

Section 4

Operators 4.3.3.2

The intrinsic binary arithmetic operators for the complex type are: `+`, `−`, `*`, `/`, and `**`. The intrinsic unary arithmetic operators are: `+` and `−`. The intrinsic relational operators are: `.EQ.`, `==`, `.NE.`, and `/=`. The arithmetic operators specify complex arithmetic; the relationals compare operands to produce default logical results. The result of an intrinsic arithmetic operation on complex operands is a complex entity. If one of the operands is an integer or real entity, the result is still a complex entity.

Form for constants 4.3.3.3

A complex literal constant is written as two literal constants that are real or integer , separated by a comma, and enclosed in parentheses, as follows:

(*real_part* **,** *imag_part*)

The CF90 compiler allows the real and imaginary portions of a complex literal constant to be named constants.

> **ANSI/ISO:** The Fortran 90 standard does not specify the use of named constants as the real or imaginary components of a complex literal constant.

The format for a *complex_literal_constant* is defined as follows:

R417	*complex_literal_constant*	**is**	(*real_part* **,** *imag_part*)
R418	*real_part*	**is**	*signed_int_literal_constant*
		or	*signed_real_literal_constant*
R419	*imag_part*	**is**	*signed_int_literal_constant*
		or	*signed_real_literal_constant*

Section 4

Examples:

```
(3.0, -3.0)
(6, -7.6E9)
(3.0_HIGH, 1.6E9_LOW)
```

A real kind parameter can be specified for either one of the two real values. If a different real kind parameter is given for each of the two real values, the complex value will have the kind parameter that specifies the greater precision, unless the kind parameters specify the same precision. If both parts are integer, each part is converted to default real. If one part is of integer type and the other is of real type, the integer value is converted to the kind and type of the real value.

Logical type
4.3.4

The name of the logical type is `LOGICAL`. A format for declaring objects to be of this type is as follows:

`LOGICAL [ ( [ KIND = ]` *kind_param* `) ] [ ,` *attribute_list* `:: ]` *entity_list*

Examples:

```
LOGICAL IR, XT
LOGICAL(KIND = SMALL), SAVE :: XMASK (3000)
```

Values
4.3.4.1

The logical data type has two values that represent true and false. The Fortran 90 standard requires processors to provide one logical kind, but the CF90 compiler provides other kinds. Each logical item occupies one word. (An object of default logical type must occupy the same unit of storage as an object of default real type.) The `KIND` intrinsic function can be used to determine the kind number of its argument. There is no intrinsic function analogous to the functions `SELECTED_INT_KIND` and `SELECTED_REAL_KIND`.

If specifying a *kind_param*, it must have one of the following values: 1, 2, 4, or 8. The default *kind_param* and the values associated with the kind values are specific to your hardware platform. These values are shown in Table 6, page 97. See the *CF90 Commands and Directives Reference Manual*, publication SR–3901, for information on changing default *kind_param* values.

Table 6. Logical kind values

kind_param	Size / Storage	Platforms
1	8 bits / 64 bits	PVP
1	8 bits / 32 bits	CRAY T3E, SPARC
2	16 bits / 64 bits	PVP
2	16 bits / 32 bits	CRAY T3E, SPARC
4	32 bits / 64 bits	PVP
4	32 bits / 32 bits	CRAY T3E
4 (default)	32 bits / 32 bits	SPARC
8 (default)	64 bits / 64 bits	PVP, CRAY T3E
8	64 bits / 64 bits	SPARC

4.3.4.2 Operators

The intrinsic binary operators for the logical type are as follows: conjunction (`.AND.`), inclusive disjunction (`.OR.`), logical equivalence (`.EQV.`), and logical nonequivalence (or exclusive disjunction) (`.NEQV.`). The intrinsic unary operation is negation (`.NOT.`). The CF90 compiler exclusive disjunction operator is `.XOR.`

ANSI/ISO: The Fortran 90 standard does not specify an exclusive disjunction operator.

Form for constants
4.3.4.3

There are only two logical literal constants. They can be followed by an underscore and a kind parameter. The format for a logical literal constant is defined as follows:

R421	*logical_literal_constant*	**is**	`.TRUE.` [_ *kind_param*]
		or	`.FALSE.` [_ *kind_param*]

If a *kind_param* is not specified, the type of the constant is default logical. Examples are as follows:

```
FALSE.
TRUE._WORD
```

If `.T.` and `.F.` are not defined operators in a compilation unit, the CF90 compiler recognizes `.T.` as an abbreviation for `.TRUE.` and `.F.` for `.FALSE.`

ANSI/ISO: The Fortran 90 standard does not specify the use of `.T.` or `.F.`.

Section 4

Character type
4.3.5

The name of the character type is `CHARACTER`. Declarations for objects of this type may take several different forms. One of these forms is as follows:

`CHARACTER [ ( [ LEN = ]` *length_parameter* `[ , [ KIND = ]` *kind_param* `] ) ]`
`[ ,` *attribute_list* `:: ]` *entity_list*

The *length_parameter* can be an asterisk or a specification expression, which is described in subsection 7.2.9.3, page 284. The various forms of the `CHARACTER` statement are described in subsection 5.1.6, page 142, but the following examples use the form specified previously:

```
CHARACTER(80) LINE
CHARACTER(*) GREETING
CHARACTER(LEN = 30, KIND = ASCII), DIMENSION(10) :: C1
```

Values
4.3.5.1

The character data type has a set of values composed of character strings. A character string is a sequence of characters, numbered from left to right 1, 2, ..., n, where n is the length of (number of characters in) the string. Both length and kind are type parameters for the character type. If no length parameter is specified, the length is 1. A character string can have length 0. The maximum length permitted for character strings is 2, 097, 151. Although the Fortran 90 standard permits a processor to provide more than one character kind, the CF90 compiler supports only one, ASCII. Thus, the CF90 compiler does not support any nondefault character kinds. If *kind_param* is specified, it must have the value 1.

The Fortran 90 standard specifies only a partial collating sequence because it is concerned only that operations that compare character objects containing only characters from the Fortran character set will be portable across different processors. Because the CF90 compiler supports the ASCII character set, it follows the Fortran 90 standard's collating requirements. The intrinsic functions `ACHAR` and `IACHAR` convert between numeric values and ASCII characters. The intrinsic functions `LGT`, `LGE`, `LLE`, and `LLT` provide comparisons between strings based on the ASCII collating sequence.

Operators
4.3.5.2

The binary operation concatenation (`//`) is the only intrinsic operation on character entities and has a character entity as a result. A number of intrinsic functions are provided that perform character operations. These are described in the *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903. The intrinsic relational operators on objects of type character are `.LT.`, `<` , `.LE.`, `<=`, `.EQ.`, `==`, `.NE.`, `/=`, `.GE.`, `>=`, `.GT.`, and `>`. The relational operations can be used to compare character entities, but because of possible processor-dependent collating sequences, care must be taken if the results are intended to be portable.

Form for constants
4.3.5.3

A character literal constant is written as a sequence of characters, enclosed either by apostrophes or quotation marks. The format for a *char_literal_constant* is as follows:

R420	*char_literal_constant*	**is**	[*kind_param* _] ' [*ASCII_char*] ... '
		or	[*kind_param* _] " [*ASCII_char*] ... "

Note that, unlike the other intrinsic types, the kind parameter for the character literal constant precedes the constant.

If the string delimiter character (either an apostrophe or a quotation mark) is required as part of the constant, two consecutive such characters with no intervening blanks serve to represent a single such character in the string.

Examples:

```
"DON'T"
'DON''T'
```

These two examples have the value `DON'T`. A zero-length character constant can be written as `""` or `''`. The quotation marks or apostrophes are immediately adjacent to each other.

Section 4

Boolean type (CF90 extension)
4.3.6

A Boolean constant represents the literal content of a single storage unit. There are no Boolean variables or arrays, and there is no Boolean type statement. A bitwise logical (masking) expression has a Boolean result, with each of its bits representing the result of one or more logical operations on the corresponding bit of the expression's operands.

ANSI/ISO: The Fortran 90 standard does not provide for Boolean values.

When an operand of a binary arithmetic or relational operator is Boolean, the operation is performed as if the Boolean operand has the same type as the other operand (that is, no data conversion occurs). If both operands are of type Boolean, the operation is performed as if they were of type integer.

No user-specified function can generate a Boolean result, but some (nonstandard) intrinsic functions can generate Boolean results.

Boolean and logical types differ in the following ways:

- Variables, arrays, and functions can be of logical type, and there is a `LOGICAL` type statement.
- A logical variable or constant represents only one value of true or false (rather than separate bit values), and a logical expression yields one true or false value.
- Logical entities are invalid in arithmetic, relational, or bitwise logical expressions, while Boolean entities are valid. (Note, however, that results of relational expressions are logical.)

A Boolean constant can be written as an octal, hexadecimal, or Hollerith value. There is no form for binary digits. Boolean constants can represent up to 256 bits of data. This size limit corresponds to the size of the largest numeric type, `COMPLEX(KIND = 16)`. The ultimate size and make-up of the constant is dependent on its context. The constant is truncated or padded to match the size of the type implied by its context. These forms use the notation described in the following subsections.

Octal form
4.3.6.1

The octal form contains 1 to 86 digits (0 through 7) in either of the following two forms:

- It can be a string of digits followed by the letter B or b, as in `177B`.
- It can be a quoted string of digits followed by the letter O or o, as in `"177"O`.
- It can be a quoted string of digits preceded by the letter O or o, as in `O"177"`.

The 86 digits in a Boolean value correspond to the internal representation of four 64-bit words or eight 32-bit words. If all 86 digits are specified, the leftmost octal digit must be only 0 or 1, representing the content of the leftmost bit position (bit 0 in the first word) of the value. Each successive octal digit specifies the contents of the next three bit positions. The last octal digit specifies the content of the rightmost three bit positions, which are bits 61, 62, and 63 of the last (fourth or eighth) word. Blanks are ignored in fixed source form. Blanks are significant in free source form for the *ddd* `B` syntax. That is, `1 777B` would not be treated as a single value in free source form, and it would most likely result in a syntax error message.

A Boolean value represented by fewer than 86 octal digits is right justified; that is, it represents the rightmost bits of 256 bits.

When context is taken into account, the value is truncated on the left if it is too large for the context type. It is padded with 0 on the left if it is too small for the context type.

> **Note:** For a literal constant, the letter `B` indicates octal digits; in an I/O format specification, the `B` descriptor indicates binary digits.

Examples:

Boolean constant	Internal representation (octal) for a 64-bit word
`0B`	`0000000000000000000000`
`77740B`	`0000000000000000077740`

Statement example:

```
I = 1357B
```


Hexadecimal form
4.3.6.2

The hexadecimal form consists of 1 to 64 hexadecimal digits consisting of `0` through `9`, `A` through `F`, or `a` through `f` in either of the following two forms:

- It can be specified as the letter `X` or `x` followed by a string of hexadecimal digits enclosed in apostrophes or quotations marks, as in `X"FFF"`.
- It can be specified with the `X` or `x` trailing the quoted string of digits, as in `"FFF"X`.

When a Boolean value contains 64 hexadecimal digits, the binary equivalents correspond to the content of each bit position in 4 64-bit words or eight 32-bit words.

A Boolean value represented by fewer than 64 hexadecimal digits is right justified; that is, it represents the rightmost bits of 256 bits.

An optional unary minus sign (−) is allowed within the quoted string.

When context is taken into account, the value is truncated on the left if it is too large for the context type. It is padded with 0 on the left if it is too small for the context type.

Examples:

Boolean constant	Internal representation (octal) for a 64-bit word
`X'ABE'`	`0000000000000000005276`
`X"-340"`	`1777777777777777776300`
`X'1 2 3'`	`0000000000000000000443`
`X'FFFFFFFFFFFFFFFF'`	`1777777777777777777777`

Statement examples:

```
J = X"28FF"
K = X'-5A'
```

Hollerith form
4.3.6.3

A Hollerith constant is type Boolean. When a character constant is used in a bitwise logical expression, the expression is evaluated as if the value were Hollerith. A Hollerith constant can have a maximum of 32 characters.

When context is taken into account, the truncation or padding depends on the type of Hollerith syntax used.

For Hollerith using the letter `H`, the value is truncated on the right if it is too large for the context type. It is padded with blanks on the right if it is too small for the context type.

For Hollerith using the letter `L`, the value is truncated on the right if it is too large for the context type. It is padded with 0 on the right if it is too small for the context type.

For Hollerith using the letter `R`, the value is truncated on the left if it is too large for the context type. It is padded with 0 on the left if it is too small for the context type.

4.3.7 Cray pointer type (CF90 extension)

A Cray pointer is a variable whose value is the address of another entity, which is called a pointee. The Cray pointer type statement declares both the pointer and its pointee.

> **ANSI/ISO:** The Fortran 90 standard does not provide for Cray pointer values. Cray pointers are declared as follows:
>
> `POINTER` (*pointer* , *pointee*)
>
> Fortran 90 pointers are declared as follows:
>
> `POINTER ::` [*object_name*]
>
> The two kinds of pointers cannot be mixed.

You can use pointers to access user-managed storage by dynamically associating variables and arrays to particular locations in a block of storage. Cray pointers do not provide convenient manipulation of linked lists because, for optimization purposes, it is assumed that no two pointers have the same value. Cray pointers also allow the accessing of absolute memory locations.

The value of a Cray pointer occupies one machine word on Cray PVP systems other than CRAY T90 systems. The value of a Cray pointer occupies 2 machine words on CRAY T90 systems, CRAY T3E systems, and SPARC systems. Its range of values depends on the size of memory for the machine in use.

Restrictions on Cray pointers are as follows:

- A Cray pointer cannot be pointed to by another Cray or Fortran 90 pointer; that is, a Cray pointer cannot also be a pointee or a target.
- A Cray pointer cannot appear in a `PARAMETER` statement or in a type declaration statement that includes the `PARAMETER` attribute.

Section 4

- A Cray pointer variable cannot be declared to be of any other data type.
- A Cray character pointer cannot appear in a `DATA` statement.
- An array of Cray pointers is not allowed.
- A Cray pointer cannot be a component of a structure.

Restrictions on Cray pointees are as follows:

- A Cray pointee cannot appear in a `SAVE`, `DATA`, `EQUIVALENCE`, `COMMON`, or `PARAMETER` statement.
- A Cray pointee cannot be a dummy argument; that is, it cannot appear in a `FUNCTION`, `SUBROUTINE`, or `ENTRY` statement.
- A function value cannot be a Cray pointee.
- A Cray pointee cannot be a structure or a structure component.
- A Cray pointee cannot be of a derived type.

> **Note:** Cray pointees can be of type character, but their Cray pointers are different from other Cray pointers; the two kinds cannot be mixed in the same expression.

The Cray pointer is a variable of type Cray pointer and can appear in a `COMMON` list or be a dummy argument in a subprogram.

The Cray pointee does not have an address until the value of the Cray pointer is defined; the pointee is stored starting at the location specified by the pointer. Any change in the value of a Cray pointer causes subsequent references to the corresponding pointer to refer to the new location.

Cray pointers can be assigned values in the following ways:

- A Cray pointer can be set as an absolute address. For example:

```
Q = 0
```

- Cray pointers can have integer expressions added to or subtracted from them and can be assigned to or from integer variables. For example:

```
P = Q + 100
```

Section 4

However , Cray pointers are not integers. For example, assigning a Cray pointer to a real variable is not allowed.

The (nonstandard) LOC intrinsic function generates the address of a variable and can be used to define a Cray pointer.

```
P = LOC(X)
```

The following example uses Cray pointers in the ways just described:

```
SUBROUTINE SUB(N)
COMMON POOL(100000), WORDS(1000)
INTEGER BLK(128), WORD64
REAL A(1000), B(N), C(100000-N-1000)
POINTER(PBLK,BLK), (IA,A), (IB,B), &
      (IC,C), (ADDRESS,WORD64)
ADDRESS = LOC(WORDS) + 64
PBLK = LOC(WORDS)
IA = LOC(POOL)
IB = IA + 1000
IC = IB + N
```

WORD64 in this example refers to the contents of absolute address 64; BLK is an array that occupies the first 128 words of memory; A is an array of length 1000 located in blank common; B follows A and is of length N; C follows B. A, B, and C are associated with POOL. On Cray PVP systems and CRAY T3E systems, WORD64 is the same as BLK(65) because BLK(1) is at the initial address of words. On SPARC systems, WORD64 is the same as BLK(17) because Cray pointers are byte addresses and the INTEGER elements of array BLK are each 4 bytes long.

If a pointee is of a data type that is one machine word or longer, the address stored in a pointer is a word address. If the pointee data type is less than one word, the address is a byte address. The following example also uses Cray pointers:

```
PROGRAM TEST
REAL X(10), Y(10), A(10)
POINTER (P,X), (Q,Y)

P = LOC(A(1))
Q = LOC(A(2))
I = P
J = Q

IF ( (J-I) .EQ. (Q-P) )  THEN
   PRINT *, 'Not a byte addressable machine'
ELSE
   PRINT *, 'Byte addressable machine'
ENDIF
END
```

On a CRAY C90 system, this prints the following:

```
Not a byte addressable machine
```

On a CRAY T3E system, this prints the following:

```
Byte addressable machine
```

For purposes of optimization, the compiler assumes that the storage of a pointee is never overlaid on the storage of another variable; that is, it assumes that a pointee is not associated with another variable or array. This kind of association occurs when a Cray pointer has two pointees, or when two Cray pointers are given the same value. Although these practices are sometimes used deliberately (such as for equivalencing arrays), results can differ depending on whether optimization is turned on or off. You are responsible for preventing such association.

```
POINTER(P,B), (P,C)
REAL X, B, C
P = LOC(X)
B = 1.0
C = 2.0
PRINT *, B
```

Because B and C have the same pointer, the assignment of 2.0 to C gives the same value to B; therefore, B will print as 2.0 even though it was assigned 1.0.

As with a variable in common storage, a pointee, pointer, or argument to a LOC intrinsic function is stored in memory before a call to an external procedure and is read out of memory at its next reference. The variable is also stored before a RETURN or END statement of a subprogram.

4.3.8 Cray character pointer type (CF90 extension)

If a pointee is declared as character type, its Cray pointer is a Cray character pointer.

Restrictions for Cray pointers also apply to Cray character pointers. In addition, the following restrictions apply:

- Character pointees must be declared, prior to the Cray pointer statement, as character type with an assumed length; for example, CHARACTER*(*) A. The length of the pointee is derived from the bit length field of the Cray pointer; this can change as the Cray pointer is associated with different objects.
- When included in an I/O statement *iolist* , a Cray character pointer is treated as an integer.
- A Cray character pointer can be used in a relational operation only with another Cray character pointer. Such an operation applies only to the character address and bit offset; the length field is not used.
- A character pointee cannot be an array.

For more information on the Cray character pointer see section 5, which begins on page 131.

Derived types

4.4

Unlike the intrinsic types that are defined by the language, you must define derived types. These types have the same utility as the intrinsic types. For example, variables of these types can be declared, passed as procedure arguments, and returned as function results.

A derived-type definition specifies a name for the type; this name is used to declare objects of the type. A derived-type definition also specifies components of the type, of which there must be at least one. A component can be of intrinsic or derived type; if it is of derived type, it can be resolved into components, called *ultimate components.* These ultimate components are of intrinsic type and can be pointers.

If the type definition appears in a module, the type definition may contain the keywords `PUBLIC` or `PRIVATE`. Generally, entities specified in a module can be kept private to the module and will not be available outside the module. This is true of data objects, module subprograms, and type definitions. By default, entities specified in a module are available to any program unit that accesses the module; that is, they have `PUBLIC` accessibility by default. This default can be changed by inserting a `PRIVATE` statement ahead of the specifications and definitions in the module. Individual entities can be specified to have either the `PUBLIC` or `PRIVATE` attribute regardless of the default. For a type definition, this can be accomplished by a `PUBLIC` or `PRIVATE` specifier in the `TYPE` statement of the type definition. The keyword `PRIVATE` can be used in two ways in type definitions in a module. One way makes the entire type private to the module; the other way allows the type name to be known outside the module, but not the names or attributes of its components. A separate `PRIVATE` statement that mentions the type name or a `PRIVATE` specifier in the `TYPE` statement of the type definition provides the first of these. An optional `PRIVATE` statement inside the type definition provides the second. See subsection 4.4.1, page 110, for examples of a private type and a public type with private components.

A type definition can contain a `SEQUENCE` statement. The Fortran 90 standard allows a processor to rearrange the components of a derived type in any convenient order. However, if a `SEQUENCE` statement appears inside the type definition, the type is considered to be a *sequence type.* In this case, the processor must allocate storage for the components in the declared order so that structures declared to be of the derived type can appear in COMMON and EQUIVALENCE statements. See subsection 4.4.1 for an example of a sequence type.

A derived type has a set of values that is every combination of the permitted values for the components of the type. The language provides a syntax for constants of the intrinsic types; it provides a somewhat similar mechanism, called a *structure constructor,* to specify a value for a derived type. A constructor can be used in a PARAMETER statement and in a type declaration statement to define derived-type named constants; they can be used in DATA statements to specify initial values; and they can be used as structure-valued operands in expressions.

User -defined functions and subroutines must be used to define operations on entities of derived type. Thus, the four properties of the intrinsic types (possession of a name, a set of values, a set of operations, and a syntactic mechanism to specify constant values) are also provided for derived types.

Derived type definition
4.4.1

A derived type definition gives a derived type a name and specifies the types and attributes of its components. A derived type definition begins with a TYPE statement, ends with an END TYPE statement, and has component declarations in between. The following example defines type PATIENT:

```
TYPE PATIENT
   INTEGER          PULSE_RATE
   REAL             TEMPERATURE
   CHARACTER*300    PROGNOSIS
END TYPE PATIENT
```

The format of a *derived_type_def* is as follows:

> TYPE [[, *access_spec*] ::] *type_name*
> [*private_sequence_stmt*] ...
> *component_def_stmt*
> [*component_def_stmt*] ...
> END TYPE [*type_name*]

A derived-type definition is defined as follows:

R422	*derived_type_def*	**is**	*derived_type_stmt* [*private_sequence_stmt*] ... *component_def_stmt* [*component_def_stmt*] ... *end_type_stmt*
R423	*private_sequence_stmt*	**is** **or**	`PRIVATE` `SEQUENCE`
R424	*derived_type_stmt*	**is**	`TYPE` [[, *access_spec*] `::`] *type_name*
R425	*end_type_stmt*	**is**	`END TYPE` [*type_name*]
R426	*component_def_stmt*	**is**	*type_spec* ≡ ≡ [[, *component_attr_spec_list*] `::`] ≡ ≡ *component_decl_list*
R427	*component_attr_spec*	**is**	`POINTER` `DIMENSION` (*component_array_spec*)
R428	*component_array_spec*	**is** **or**	*explicit_shape_spec_list* *deferred_shape_spec_list*
R429	*component_decl*	**is**	*component_name* ≡ ≡ [(*component_array_spec*)] ≡ ≡ [* *char_length*]

For *access_spec* , specify either `PRIVATE` or `PUBLIC`.

The *component_array_spec* must be a deferred-shape array if the `POINTER` attribute is present; otherwise, it must be an explicit-shape array.

The name of the derived type must not be the same as any intrinsic type or locally accessible name in the same class; it has the scope of local names declared in the scoping unit, which means that it is accessible by use or host association in other scoping units. A component name has the scope of the

Section 4

type definition only; another type definition in the same scoping unit may specify the same component name. For more information on local entities and scope, see the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–2903.

If the `END TYPE` statement is followed by a name, it must be the name specified in the `TYPE` statement.

A type can be defined only once within a scoping unit.

A `PRIVATE` statement must not appear more than once in a type definition.

A `SEQUENCE` statement must not appear more than once in a type definition

If `SEQUENCE` is present, all derived types specified as components must also be sequence types.

The keywords `PUBLIC` and `PRIVATE` can appear only if the definition is in the specification part of a module.

There must be at least one component definition statement in a type definition.

No component attribute can appear more than once in a specified component definition statement.

A component can be declared to have the same type as the type being defined only if it has the `POINTER` attribute.

An array component without the `POINTER` attribute must be specified with an explicit-shape specification where the bounds are integer constant expressions.

If a component is of type character with a specified length, the length must be an integer constant specification expression. If the length is not specified, it is 1.

The following example shows a derived-type definition with four components (three integer and one character):

```
TYPE COLOR
   INTEGER :: HUE, SHADE, INTENSITY
   CHARACTER(LEN = 30) :: NAME
END TYPE COLOR
```

The following is a format for declaring variables of derived type:

`TYPE (` *type_name* `) [ ,` *attribute_list* `:: ]` *entity_list*

For example, variables of type `COLOR` can be declared as follows:

```
TYPE(COLOR) MY_FAVORITE
TYPE(COLOR) RAINBOW(7)
TYPE(COLOR), DIMENSION (100) :: SELECTIONS
```

The object `MY_FAVORITE` is a structure. The objects `RAINBOW` and `SELECTIONS` are arrays of structures.

Note that the initial statement of a type definition and the statement used to declare objects of derived type both begin with the keyword `TYPE`. The initial statement of a type definition is called a *derived-type statement,* and the statement used to declare objects of derived type is called a `TYPE` statement . The type name in a derived-type statement is not enclosed in parentheses, whereas the type name in a `TYPE` statement is.

A component of a structure is referenced using a percent sign, as in the following template:

parent_structure `%` *component_name*

Example:

```
MY_FAVORITE % HUE
RAINBOW(3) % NAME
```

The following examples show definitions of derived types. Each example illustrates a different aspect of a type definition:

- A derived type with a component of a different derived type
- A derived type with a pointer component
- A derived type with a pointer component of the type being defined

Section 4

- A private type definition
- A public type definition with private components

Example 1: A derived type can have a component that is of a different derived type. Type WEATHER in the following example has a component of type TEMPERATURES.

```
TYPE TEMPERATURES
   INTEGER :: HIGH, LOW
END TYPE TEMPERATURES

TYPE WEATHER
   CHARACTER(LEN = 32) :: CITY
   TYPE(TEMPERATURES) :: RANGE(1950:2050)
END TYPE WEATHER

TYPE(WEATHER) WORLDWIDE(200)
```

WORLDWIDE is an array of type WEATHER. Components of an element of the array are referenced as follows:

```
WORLDWIDE(I)%CITY = "Nome"
WORLDWIDE(I)%RANGE(1990)%LOW = -83
```

Example 2: A derived type can have a component that is a pointer, as follows:

```
TYPE ABSTRACT
   CHARACTER(LEN = 50) TITLE
   INTEGER NUM_OF_PAGES
   CHARACTER, POINTER :: TEXT(:)
END TYPE ABSTRACT
```

Any object of type ABSTRACT will have three components: TITLE, NUM_OF_PAGES, and TEXT. TEXT is a pointer to an array of character strings, each of which is of length one. The array size is determined during program execution. The space for the target of TEXT can be allocated, or TEXT can be pointer-assigned to existing space. For information on the ALLOCATE statement, see subsection 6.5.1, page 225. For information on pointer assignment, see subsection 7.5.3, page 313.

Example 3: A derived type can have a pointer component that is of the type being defined. This is useful in creating linked lists and trees, as follows:

```
TYPE LINK
   REAL VALUE
   TYPE(LINK), POINTER :: PREVIOUS
   TYPE(LINK), POINTER :: NEXT
END TYPE LINK
```

Example 4: A type definition in a module can be kept private to the module, as follows:

```
TYPE, PRIVATE :: FILE
   INTEGER DRAWER_NO
   CHARACTER(LEN = 20) FOLDER_NAME
   CHARACTER(LEN = 5) ACCESS_LEVEL
END TYPE FILE
```

When a module that contains this type definition is accessed by another scoping unit, the type `FILE` is not available.

Example 5: A type definition can be public while its components are kept private, as follows:

```
MODULE COORDINATES
   TYPE POINT
      PRIVATE
      REAL X, Y
   END TYPE POINT
      ...
END MODULE COORDINATES
```

In a program unit that uses module `COORDINATES`, variables of type `POINT` can be declared; values of type `POINT` can be passed as arguments; and if the program unit is a function, a value of type `POINT` can be returned as the result. However, the internal structure of the type (its components) is not available. If the type `POINT` is changed to the following, no other program unit that uses `COORDINATES` will need to be changed:

```
TYPE POINT
   PRIVATE
   REAL RHO, THETA
END TYPE POINT
```

If a subprogram dummy argument is of derived type, the corresponding actual argument must be of the same type. There are two ways in which objects in different scoping units can be declared to be of the same type. Two data entities have the same type if they are declared with reference to the same type definition. The definition can appear in a module that is accessed or, in the case of an internal or module procedure, in the host scoping unit.

```
MODULE SHOP
   TYPE COMPONENT
      CHARACTER(LEN = 20) NAME
      INTEGER CATALOG_NUM
      REAL WEIGHT
   END TYPE COMPONENT
   TYPE(COMPONENT) PARTS(100)
CONTAINS
   SUBROUTINE GET_PART(PART, NAME)
      TYPE(COMPONENT) PART
      CHARACTER(LEN = *) NAME
      DO I = 1, 100
         IF (NAME .EQ. PARTS(I)%NAME) THEN
            PART = PARTS(I)
            RETURN
         END IF
      END DO
      PRINT *, "Part not available"
      PART%NAME = "none"
      PART%CATALOG_NUM = 0
      PART%WEIGHT = 0.0
   END SUBROUTINE GET_PART
      . . .
END MODULE SHOP

PROGRAM BUILD_MACHINE
   USE SHOP
   TYPE(COMPONENT) MOTOR(20)
   TOTAL_WEIGHT = 0.0
   CALL GET_PART(MOTOR(1), "VALVE")
   IF (MOTOR(1)%WEIGHT .NE. 0) THEN
      TOTAL_WEIGHT = TOTAL_WEIGHT + MOTOR(1)%WEIGHT
   ELSE
      . . .
   ENDIF
   . . .
END PROGRAM BUILD_MACHINE
```

Module procedure GET_PART has access to the type COMPONENT because the type definition appears in its host. Program BUILD_MACHINE has access to the type because it uses module SHOP. This allows a variable of the type, such as MOTOR(1), to be passed as an actual argument.

The other way to declare data entities in different scoping units to be of the same type is provided for programmers who choose not to use a module. Instead of a single type definition in the module, a sequence type can be defined in each of the scoping units that need access to the type. Each of the type definitions must specify the same name; the SEQUENCE property; have no private components; and have components that agree in order, name, and attributes. If this is the case, data entities declared in any of these scoping units to be of the named type are considered to be of the same type. In the following, program BUILD_MACHINE is restated to illustrate the differences between the two ways:

```
PROGRAM BUILD_MACHINE
   TYPE COMPONENT
      SEQUENCE
      CHARACTER(LEN = 20) NAME
      INTEGER CATALOG_NUM
      REAL WEIGHT
   END TYPE COMPONENT
   TYPE(COMPONENT) PARTS, MOTOR(20)
   COMMON /WAREHOUSE/ PARTS(100)
   TOTAL_WEIGHT = 0.0
   CALL GET_PART(MOTOR(1), "VALVE")
   IF (MOTOR(1)%WEIGHT .NE. 0) THEN
      TOTAL_WEIGHT = TOTAL_WEIGHT + MOTOR(1)%WEIGHT
   ELSE
      . . .
   ENDIF
      . . .
END PROGRAM BUILD_MACHINE
SUBROUTINE GET_PART(PART, NAME)
   TYPE COMPONENT
      SEQUENCE
      CHARACTER(LEN = 20) NAME
      INTEGER CATALOG_NUM
      REAL WEIGHT
   END TYPE COMPONENT
   TYPE(COMPONENT) PART, PARTS
   CHARACTER(LEN = *) NAME
   COMMON /WAREHOUSE/ PARTS(100)
   DO I = 1, 100
      IF (NAME .EQ. PARTS(I)%NAME) THEN
         PART = PARTS(I)
         RETURN
      END IF
   END DO
   PART%NAME = "none"
   PART%CATALOG_NUM = 0
   PART%WEIGHT = 0.0
   PRINT *, "Part not available"
END SUBROUTINE GET_PART
   . . .
```

Section 4

In this example, type COMPONENT in program BUILD_MACHINE and type COMPONENT in subroutine GET_PART are the same because they are sequence types with the same name; have no private components; and have components that agree in order, name, and attributes. This allows variables of the type to appear in COMMON and be passed as arguments. Note that this example is less concise, particularly if more procedures need to access the type definition, and therefore may be more error prone than the previous example.

Type COMPONENT is a sequence type because its definition contains a SEQUENCE statement. If all of the ultimate components of a sequence type are of type default integer, default real, double-precision real, default complex, or default logical, and are not pointers, the type is a *numeric sequence type.* An object of numeric sequence type can be equivalenced to default numeric objects.

If all of the ultimate components of a sequence type are of type character and are not pointers, the type is a *character sequence type.* An object of character sequence type may be equivalenced to character objects.

Derived type values
4.4.2

The set of values of a derived type consists of all combinations of the possibilities for component values that are consistent with the components specified in the type definition.

Derived type operations
4.4.3

Any operation involving a derived-type entity must be defined explicitly by a function with an OPERATOR interface. Assignment, other than the intrinsic assignment provided for entities of the same derived type, must be defined by a subroutine with an ASSIGNMENT interface. See the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903, for a description.

Suppose it is desirable to determine the number of words and lines in a section of text. The information is available for each paragraph. A type named PARAGRAPH is defined as follows:

```
TYPE PARAGRAPH
   INTEGER NUM_WORDS, NUM_LINES
   CHARACTER(LEN = 30) SUBJECT
END TYPE PARAGRAPH
```

Suppose that it is now desirable to define an operator for adding the counts associated with the paragraphs. The following OPERATOR interface is required for the function that defines the addition operation for objects of type PARAGRAPH:

```
INTERFACE OPERATOR (+)
   MODULE PROCEDURE ADDP
END INTERFACE
```

The following definition of addition for objects of type PARAGRAPH adds the words and lines, but it does nothing with the component SUBJECT because that would have no useful meaning.

```
TYPE(PARAGRAPH) FUNCTION ADDP(P1, P2)
   TYPE(PARAGRAPH) P1, P2
   INTENT(IN) P1, P2
   ADDP%NUM_WORDS = P1%NUM_WORDS + P2%NUM_WORDS
   ADDP%NUM_LINES = P1%NUM_LINES + P2%NUM_LINES
END FUNCTION ADDP
```

If the following variables were declared, the expression BIRDS+BEES would be defined and could be evaluated in the module subprogram as well as any program unit accessing the module:

```
TYPE(PARAGRAPH) BIRDS, BEES
```


Syntax for specifying derived-type constant expressions
4.4.4

When a derived type is defined, a structure constructor for that type is defined automatically. The structure constructor is used to specify values of the type. It specifies a sequence of values, one for each of the components of the type. A structure constructor whose values are all constant expressions is a derived-type constant expression. (This is why a derived-type value is formed by a constructor. There is no such thing as a structure constant; there are only structure constructors, some of which may be constant expressions.) A named constant of user-defined type can be assigned such a value. Structure constructors are described in subsection 4.5.

A component of a derived type can be an array. In this case a mechanism called an array constructor is used to specify that component of the type. Array constructors are described in subsection 4.6, page 125, and a general discussion of arrays can be found in subsection 6.4, page 212.

4.5 Structure constructors

A *structure constructor* is a mechanism that is used to specify a value of a derived type by specifying a sequence of values for the components of the type. If a component is of derived type, an embedded structure constructor is required to specify the value of that component. If a component is an array, an embedded array constructor is required to specify the values for that component.

A structure constructor is the name of the type followed by a sequence of component values in parentheses. For example, a value of type `COLOR` (from subsection 4.4.1, page 110) can be constructed with the following structure constructor:

```
COLOR(I, J, K, "MAGENTA")
```

For information on derived types, see subsection 4.4.1, page 110.

The format for a *structure_constructor* is defined as follows:

R430	*structure_constructor*	**is**	*type_name* (*expr_list*)

There must be a value in the expression list for each component.

The expressions must agree in number and order with the components of the derived type. Values may be converted (in the same way they would be for an assignment statement) to agree in type, kind, length, and, in some cases, rank, with the components. The conversions permitted are those for intrinsic assignment, in which the component is the variable on the left and the expression is the one given in the structure constructor corresponding to the component. Rank must be conformable according to the rules of assignment conformance. That is, the shapes must conform or the expression can be a scalar broadcast to an array component.

If a component is an explicit-shape array (that is, a nonpointer array), the array constructor for it in the expression list must be the same shape as the component.

If a component is a pointer, the value for it in the expression list must evaluate to an allowable target for the pointer. A constant is not an allowable target.

A structure constructor must not appear before that type is defined.

The structure constructor for a private type or a public type with private components is not available outside the module in which the type is defined.

If the values in a structure constructor are constants, you can use the structure constructor to specify a named constant, as in the following example:

```
PARAMETER( TEAL = COLOR(14, 7, 3, ”TEAL”) )
TYPE(COMPONENT), PARAMETER :: NO_PART = COMPONENT(”none”, 0, 0.0) )
```

Following are several examples of structure constructors for types with somewhat different components:

- A type with a component that is of derived type
- A type with an array component
- A type with a pointer component

Example 1: A structure constructor for a type that has a derived type as a component must provide a value for each of the components. A component may be of derived type, in which case a structure constructor is required for the component. In the following example, type `RING` has a component of type `STONE`:

```
TYPE STONE
    REAL            CARETS
    INTEGER         SHAPE
    CHARACTER(30)   NAME
END TYPE STONE

TYPE RING
    REAL            EST_VALUE
    CHARACTER(30)   INSURER
    TYPE (STONE)    JEWEL
END TYPE RING
```

If OVAL is a named integer constant, an example of a structure constructor for a value of type RING is as follows:

```
RING (5000.00, ”Lloyds”, STONE(2.5, OVAL, ”emerald”) )
```

Example 2: If a type is specified with an array component, the value that corresponds to the array component in the expression list of the structure constructor must conform with the shape of the array component. For example, type ORCHARD has an array component as follows:

```
TYPE ORCHARD
    INTEGER                AGE, NUM_TREES
    CHARACTER(LEN = 20)    VARIETY(10)
END TYPE
```

Assume the following declarations:

```
CHARACTER(LEN = 20) CATALOG(16, 12)
PARAMETER(LEMON = 3)
```

A structure constructor for a value of type ORCHARD is as follows:

```
ORCHARD (5, ROWS * NUM_PER_ROW, CATALOG(LEMON, 1:10) )
```

Example 3: When a component of the type is a pointer, the corresponding structure constructor expression must evaluate to an object that would be an allowable target for such a pointer in a pointer assignment statement. Assume that the variable SYNOPSIS is declared as follows:

```
CHARACTER, TARGET :: SYNOPSIS(4000)
```

The following value of the type ABSTRACT (from subsection 4.4.1, page 110) can then be constructed:

```
ABSTRACT("War and Peace", 1025, SYNOPSIS)
```

A constant expression cannot be constructed for a type with a pointer component because a constant is not an allowable target in a pointer assignment statement.

Array constructors

4.6

An array constructor is used to specify the value of an array. More precisely, an *array constructor* is a mechanism that is used to specify a sequence of scalar values that is interpreted as a rank-one array. Syntactically, it is a sequence of scalar values and implied-DO specifications enclosed in parentheses and slashes. As with structures, there is no such thing as an array constant; there are only array constructors, some of which may be constant expressions.

```
REAL VECTOR_X(3), VECTOR_Y(2), RESULT(100)
   . . .
RESULT(1:8) = (/ 1.3, 5.6, VECTOR_X, 2.35, VECTOR_Y /)
```

The value of the first eight elements of RESULT is constructed from the values of VECTOR_X and VECTOR_Y and three real constants in the specified order . If a rank-two or greater array appears in the value list, the values of its elements are taken in array element order. If it is necessary to construct an array of rank greater than one, the RESHAPE intrinsic function can be applied to an array constructor.

The format for an *array_constructor* is as follows:

R431	*array_constructor*	**is**	`(/` *ac_value_list* `/)`
R432	*ac_value*	**is**	*expr*
		or	*ac_implied_do*
R433	*ac_implied_do*	**is**	`(` *ac_value_list* `,` *ac_implied_do_control* `)`
R434	*ac_implied_do_control*	**is**	*ac_do_variable* `=` *scalar_int_expr* `,` ≡
			≡ *scalar_int_expr* [`,` *scalar_int_expr*]
R435	*ac_do_variable*	**is**	*scalar_int_variable*

Each *ac_value* expression in the array constructor must have the same type, type parameters, and length parameters. In particular, this means that if each *ac_value* is a character literal constant, each constant must have the same length.

The type and type parameters of an array constructor are those of its *ac_value* expressions. The *expr* is defined in R723.

If the *ac_implied_do* yields no values, the array is a rank one, zero-sized array .

An *ac_do_variable* must be a scalar integer named variable. This variable has the scope of this *ac_implied_do* .

If an *ac_implied_do* is contained within another *ac_implied_do* , they must not have the same *ac_do_variable* .

Three possibilities for an *ac_value* are as follows:

1. It can be a scalar expression, as is each *ac_value* in the following:

```
(/ 1.2, 3.5, 1.1 /)
```

2. It can be an array expression, as is each *ac_value* in the following:

```
(/ A(I, 1:3), A(I+1, 6:8) /)
```

Section 4

3. It can be an implied-`DO` specification, as in the following:

```
(/ (SQRT(REAL(I)), I = 1, 9) /)
```

The possibilities can be mixed in a single array constructor, as follows:

```
(/ 1.2, B(2:6,:), (REAL(I), I = 1, N), 3.5 /)
```

If an *ac_value* is an array expression, the values of the elements of the expression in array element order become the values of the array constructor . For example, the values that result from the example in possibility 2 are as follows:

```
(/ A(I,1), A(I,2), A(I,3), A(I+1,6), A(I+1,7), A(I+1,8) /)
```

For more information on array element order, see subsection 6.4.7, page 222.

If an *ac_value* is an implied-`DO` specification, it is expanded to form a sequence of values under control of the *ac_do_variable* as in the `DO` construct. For example, the values that result from the example in possibility 3 are as follows:

```
(/1.0, 1.414, 1.732, 2.0, 2.236, 2.449, 2.645, 2.828, 3.0/)
```

For more information on the `DO` construct, see subsection 8.5, page 336.

If every expression in an array constructor is a constant expression, the array constructor is a constant expression as in the example above. Such an array constructor can be used to assign a value to a named constant.

```
REAL X(3), EXTENDED_X(4)
PARAMETER(X = (/ 2.0, 4.0, 6.0 /) )
REAL, PARAMETER :: EXTENDED_X = (/ 0.0, X /) )
```

The following are examples of array constructors:

1. A constructor for a rank two array
2. A constructor for an array of derived type
3. A constructor for a value of derived type with an array component
4. A constructor for a value of derived type with a rank two array component

Example 1: To create a value for an array of rank greater than one, the RESHAPE intrinsic function must be used. With this function, a one-dimensional array may be reshaped into any allowable array shape.

```
Y = RESHAPE(SOURCE = (/ 2.0, (/ 4.5, 4.0 /), Z /),  &
      SHAPE = (/ 3, 2 /))
```

If Z has the value given in possibility 1, then Y is a 3 by 2 array with the following elements:

2.0	1.2
4.5	3.5
4.0	1.1

Example 2: It might be necessary to construct an array value of derived type.

```
TYPE PERSON
   INTEGER AGE
   CHARACTER(LEN = 40) NAME
END TYPE PERSON

TYPE(PERSON) CAR_POOL(3)

CAR_POOL = (/ PERSON(35, "SCHMITT"),  &
      PERSON(57, "LOPEZ"), PERSON(26, "YUNG") /)
```

Example 3: When one of the components of a derived type is an array, then an array constructor must be used in the structure constructor for the derived type. Suppose that the definition for type COLOR is as follows, which differs slightly from that stated previously:

```
TYPE COLOR
   INTEGER PROPERTIES(3)
   CHARACTER(LEN = 30) NAME
END TYPE COLOR
```

The following value of the revised type COLOR can be constructed:

```
COLOR((/ 5, 20, 8 /), "MAGENTA")
```

Example 4: A derived type might contain an array of rank two or greater, as follows:

```
TYPE LINE
   REAL      COORD(2, 2)
   REAL      WIDTH
   INTEGER   PATTERN
END TYPE LINE
```

The values of COORD are the coordinates x_1, y_1 and x_2, y_2 representing the end points of a line; WIDTH is the line width in centimeters; and PATTERN is 1 for a solid line, 2 for a dashed line, and 3 for a dotted line. An object of type LINE is declared and given a value as follows:

```
TYPE(LINE) SLOPE
   . . .
SLOPE = LINE(RESHAPE((/ 0.0, 1.0, 0.0, 2.0 /), (/ 2, 2 /)), 0.1, 1)
```

The RESHAPE intrinsic function is used to construct a value that represents a solid line from (0,0) to (1,2) of width 0.1 centimeters.

Section 4

Declarations [5]

Declarations are used to specify the type and other attributes of program entities. The attributes that an entity possesses determine how the entity can be used in a program. Every variable and function has a type, which is the most important of the attributes; type is discussed in section 4, which begins on page 73. However, type is only one of a number of attributes that an entity may possess. Some entities, such as subroutines and namelist groups, do not have a type but may possess other attributes. In addition, there are relationships among objects that can be specified by `EQUIVALENCE`, `COMMON`, and `NAMELIST` statements. Declarations are used to specify these attributes and relationships.

Generally , Fortran keywords are used to declare the attributes for an entity. The following list summarizes these keywords:

Attribute	Keyword
Type	`INTEGER`, `REAL` (and `DOUBLE PRECISION`), `COMPLEX`, `LOGICAL`, `CHARACTER`, `TYPE` (user-defined name)
Array properties	`DIMENSION`, `ALLOCATABLE`
Pointer properties	`POINTER`, `TARGET`
Setting values	`DATA`, `PARAMETER`
Object accessibility and use	`PUBLIC`, `PRIVATE`, `INTENT`, `OPTIONAL`, `SAVE`
Procedure properties	`EXTERNAL`, `INTRINSIC`

The attributes are described and illustrated in turn using either of the two forms that attribute specifications can take: entity-oriented and attribute-oriented.

In FORTRAN 77, it was necessary to use a different statement for each attribute given to a variable or a collection of variables. For example:

```
INTEGER A, B, C
SAVE    A, B, C
```

In Fortran 90, for objects that have a type, the other attributes may be included in the type declaration statement. For example:

```
INTEGER, SAVE :: A, B, C
```

Collecting the attributes into a single statement is sometimes more convenient for readers of programs. It eliminates searching through many declaration statements to locate all attributes of a particular object. Emphasis can be placed on an object and its attributes (entity-oriented declaration) or on an attribute and the objects that possess the attribute (attribute-oriented declaration), whichever is preferred by a programmer. In both forms, dimensionality can be specified as an attribute or as an attachment to the object name.

The following are examples of entity-oriented declaration statements:

```
REAL, DIMENSION(20), SAVE :: X
```

or

```
REAL, SAVE :: X(20)
```

The following are examples of attribute-oriented declaration statements:

```
REAL X
DIMENSION X(20)
SAVE X
```

or

```
REAL X(20)
SAVE X
```

If attributes are not declared for a data object, defaults apply. Generally, if an attribute is not specified for an object, it is assumed that the object does not possess the attribute. However, each data object has a type, and if this is not explicitly specified, it is assumed from the first letter of its name. You can use the `IMPLICIT` statement to specify any intrinsic or user-defined type for an initial letter or a range of initial letters. The `IMPLICIT NONE` statement, on the other hand, removes implicit typing and thus requires explicit type declarations for every named data object in the scoping unit.

Fortran 90 provides new dynamic data objects that can be sized at the time a program is executed. These include allocatable arrays and objects with the `POINTER` attribute. They also include automatic data objects (arrays of any type and character strings) that are created on entry into a procedure. Only objects whose size may vary are called automatic.

Other declarations (`NAMELIST`, `EQUIVALENCE`, and `COMMON`) establish relationships among data objects. The `NAMELIST` statement is used to name a collection of objects so that they can be referenced by a single name in an input/output (I/O) statement. `EQUIVALENCE` provides references to storage by more than one name. `COMMON` provides a mechanism to share storage among the different units of a program.

Type declaration statements

5.1

A type declaration type statement begins with the name of the type, optionally lists other attributes, then ends with a list of variables that possess these attributes. In addition, a type declaration statement may include an initial value for a variable. If the `PARAMETER` attribute is specified on a type statement, the statement must include the value of the named constant.

The *type_declaration_stmt* is defined as follows:

R501	*type_declaration_stmt*	**is**	*type_spec* [[, *attr_spec*] ... **: :**] *entity_decl_list*
R502	*type_spec*	**is**	`INTEGER` [*kind_selector*]
CRI		**or**	`INTEGER*`*length_value*
		or	`REAL` [*kind_selector*]
CRI		**or**	`REAL*`*length_value*
		or	`DOUBLE PRECISION`
CRI		**or**	`DOUBLE PRECISION*`*length_value*
		or	`COMPLEX` [*kind_selector*]
CRI		**or**	`COMPLEX*`*length_value*
		or	`CHARACTER` [*char_selector*]
		or	`LOGICAL` [*kind_selector*]
CRI		**or**	`LOGICAL*`*length_value*
		or	`TYPE (` *type_name* `)`

CRI		**or**	`POINTER` (*pointer_name* , *pointee_name* ≡
			≡ [(*array_spec*)]) ≡
			≡ [, (*pointer_name* , *pointee_name* ≡
			≡ [(*array_spec*)])] ...
R503	*attr_spec*	**is**	`PARAMETER`
		or	*access_spec*
		or	`ALLOCATABLE`
		or	`DIMENSION` (*array_spec*)
		or	`EXTERNAL`
		or	`INTENT` (*intent_spec*)
		or	`INTRINSIC`
		or	`OPTIONAL`
		or	`POINTER`
		or	`SAVE`
		or	`TARGET`
R510	*access_spec*	**is**	`PUBLIC`
		or	`PRIVATE`
R504	*entity_decl*	**is**	*object_name* [(*array_spec*)] ≡
		or	*function_name* [* *char_length*]
			≡ [* *char_length*] [= *initialization_expr*]
R505	*kind_selector*	**is**	([`KIND` =] *scalar_int_initialization_expr*)
R732	*int_initialization_expr*	**is**	*int_expr*
R728	*int_expr*	**is**	*expr*

The double colon symbol (`::`) is required in a type declaration statement only when the type declaration statement contains two or more attributes or when it contains an *initialization_expr*.

The type specification can override or confirm the implicit type indicated by the first letter of the entity name according to the implicit typing rules in effect.

The same attribute must not appear more than once in a given type declaration statement.

An entity must not be assigned any attribute more than once in a scoping unit.

The value specified in a kind selector must be a kind type parameter allowed for that type by the CF90 compiler.

The character length option can appear only when the type specification is `CHARACTER`.

An initialization expression must be included if the `PARAMETER` attribute is specified.

A function name must be the name of an external function, an intrinsic function, a function dummy procedure, or a statement function.

An array function result name must be specified as an explicit-shape array unless it has the `POINTER` attribute, in which case it must be specified as a deferred-shape array. For information on array properties, see subsection 5.3, page 151. Note that when declaring the result of a function (inside the function), use the first form of R504. When declaring a function name (in the calling program unit), use the second form of R504.

Other rules and restrictions pertain to particular attributes; these are covered in the subsections describing those attributes. The attributes that can be used with the attribute being described are also listed. The simple forms that appear in the following subsections to illustrate attribute specification in a type declaration statement show the attribute being described first in the attribute list, but attributes can appear in any order. If these simple forms are used to construct statements, the statements will be correct, but other variations are permitted.

The following examples show type declaration statements:

```
REAL A(10)
LOGICAL, DIMENSION(5,5) :: MASK_1, MASK_2
COMPLEX :: CUBE_ROOT = (-0.5, 0.867)
INTEGER, PARAMETER :: SHORT = SELECTED_INT_KIND(4)
INTEGER(SHORT) :: K           ! Range of -9999 to 9999
REAL, ALLOCATABLE :: A1(:, :), A2(:, :, :)
TYPE(PERSON) CHAIRMAN
TYPE(NODE), POINTER :: HEAD_OF_CHAIN, END_OF_CHAIN
REAL, INTENT(IN) :: ARG1
REAL, INTRINSIC :: SIN
```

Integer
5.1.1

An `INTEGER` statement declares the names of entities to be of type integer. If a kind selector is present, it specifies the representation method. For more information on integer type, see subsection 4.3.1, page 83.

The CF90 compiler supports the following formats for declaring objects of this type:

`INTEGER [ ( [ KIND = ] `*kind_param*` ) ] [ , `*attribute_list*` :: ] `*entity_list*

`INTEGER * `*length_value*` [ , `*attribute_list*` :: ] `*entity_list*

For *kind_param* values, see subsection 4.3.1, page 83. The *length_value* values correspond to the *kind_param* values and are as follows:

- On Cray PVP systems, the values are as follows: 1, 2, 4, 6 (default) (obsolescent), and 8.
- On CRAY T3E systems, the values are as follows: 1, 2, 4, and 8 (default).
- On SPARC systems, the values are as follows: 1, 2, 4 (default), 6 (obsolescent), and 8.

Warning: Specifying `KIND=6` or `INTEGER*6` is obsolescent. When specified, a warning message is generated. On SPARC systems, when `KIND=6` is specified, `KIND=4` is used. The `KIND=6` kind type parameter will be removed in the CF90 3.0 release.

The following are examples of entity-oriented declaration statements:

```
INTEGER, DIMENSION(:), POINTER :: MILES, HOURS
INTEGER(SHORT), POINTER :: RATE, INDEX
```

Section 5

The following are examples of attribute-oriented declaration statements:

```
INTEGER :: MILES, HOURS
INTEGER(SHORT) :: RATE, INDEX
DIMENSION :: MILES(:), HOURS(:)
POINTER :: MILES, HOURS, RATE, INDEX
```

ANSI/ISO: The Fortran 90 standard does not specify the `INTEGER *` *length_value* syntax. The CF90 compiler supports this syntax to be compatible with the CF77 compiling system and other FORTRAN 77 compilers. It is recommended that this syntax not be used in any new code.

Real
5.1.2

A `REAL` statement declares the names of entities to be of type real. If a kind selector is present, it specifies the representation method. For more information on real type, see subsection 4.3.2, page 88.

The CF90 compiler supports the following formats for declaring objects of this type:

`REAL [ ( [ KIND = ]` *kind_param* `) ] [ ,` *attribute_list* `:: ]` *entity_list*

`REAL *` *length_value* `[ ,` *attribute_list* `:: ]` *entity_list*

For *kind_param* values, see subsection 4.3.2, page 88. The *length_value* values correspond to the *kind_param* values and are as follows:

- On Cray PVP systems, the values are as follows: 4, 8 (default), and 16.
- On CRAY T3E systems, the values are as follows: 4, 8 (default), and 16 (deferred).
- On SPARC systems, the values are as follows: 4 (default), 8, and 16.

Section 5

The following examples show entity-oriented declaration statements:

```
REAL(KIND = HIGH), OPTIONAL :: VARIANCE
REAL, SAVE :: A1(10, 10), A2(100, 10, 10)
```

The following examples show attribute-oriented declaration statements:

```
REAL(KIND = HIGH) VARIANCE
REAL A1(10, 10), A2(100, 10, 10)
OPTIONAL VARIANCE
SAVE A1, A2
```

ANSI/ISO: The Fortran 90 standard does not specify the `REAL *` *length_value* syntax. The CF90 compiler supports this syntax to be compatible with the CF77 compiling system and other FORTRAN 77 compilers. It is recommended that this syntax not be used in any new code.

Double precision
5.1.3

A `DOUBLE PRECISION` statement declares the names of entities to be of type real with a representation method that represents more precision than the default real representation. `DOUBLE PRECISION` is not needed in Fortran 90 because `REAL` with the appropriate kind parameter value is equivalent. A kind selector is not permitted in the `DOUBLE PRECISION` statement. For more information on type real, see subsection 4.3.2, page 88.

The CF90 compiler supports the following formats for declaring objects of this type:

`DOUBLE PRECISION [ ,` *attribute_list* `:: ]` *entity_list*

`DOUBLE PRECISION * 16 [ ,` *attribute_list* `:: ]` *entity_list*

The following examples show entity-oriented declaration statements:

```
DOUBLE PRECISION, DIMENSION(N,N) :: MATRIX_A, MATRIX_B
DOUBLE PRECISION, POINTER :: C, D, E, F(:, :)
```

The following examples show attribute-oriented declaration statements:

```
DOUBLE PRECISION :: MATRIX_A, MATRIX_B, C, D, E, F
DIMENSION :: MATRIX_A(N, N), MATRIX_B(N, N), F(:, :)
POINTER :: C, D, E, F
```

If DOUBLE is a named integer constant that has the value of the kind parameter of the double-precision real type on the target platform, the preceding entity-oriented declaration statements could be written as follows:

```
REAL (DOUBLE), DIMENSION (N,N) :: MATRIX_A, MATRIX_B
REAL (DOUBLE), POINTER :: C, D, E, F(:,:)
```

ANSI/ISO: The Fortran 90 standard does not specify the DOUBLE PRECISION * 16 syntax. The CF90 compiler supports this syntax to be compatible with the CF77 compiling system and other FORTRAN 77 compilers. It is recommended that this syntax not be used in any new code.

Note: The implementation of DOUBLE PRECISION and DOUBLE PRECISION * 16 syntax is deferred on CRAY T3E systems.

Complex

5.1.4

A COMPLEX statement declares the names of entities to be of type complex. If a kind selector is present, it specifies the representation method. For more information on complex type, see subsection 4.3.3, page 93.

The CF90 compiler supports the following formats for declaring objects of this type:

COMPLEX [([KIND =] *kind_param*)] [, *attribute_list* ::] *entity_list*

COMPLEX * *length_value* [, *attribute_list* ::] *entity_list*

Section 5

For *kind_param* values, see subsection 4.3.3, page 93. The *length_value* values correspond to the *kind_param* values in the following manner:

- On Cray PVP systems, the values are as follows:

kind_param	*length_value*
`4`	`8`
`8` (default)	`16` (default)
`16`	`32`

- On CRAY T3E systems, the values are as follows:

kind_param	*length_value*
`4`	`8`
`8` (default)	`16` (default)
`16` (deferred)	`32` (deferred)

- On SPARC systems, the values are as follows:

kind_param	*length_value*
`4` (default)	`8` (default)
`8`	`16`
`16`	`32`

The following examples show entity-oriented declaration statements:

```
COMPLEX(KIND = LOW), POINTER :: ROOTS(:)
COMPLEX, POINTER :: DISCRIMINANT, COEFFICIENTS(:)
```

The following examples show attribute-oriented declaration statements:

```
COMPLEX(KIND = LOW) :: ROOTS(:)
COMPLEX :: DISCRIMINANT, COEFFICIENTS(:)
POINTER :: ROOTS, DISCRIMINANT, COEFFICIENTS
```

ANSI/ISO: The Fortran 90 standard does not specify the `COMPLEX *` *length_value* syntax. The CF90 compiler supports this syntax to be compatible with the CF77 compiling system and other FORTRAN 77 compilers. It is recommended that this syntax not be used in any new code.

Logical
5.1.5

A `LOGICAL` statement declares the names of entities to be of type logical. If a kind selector is present, it specifies the representation method. For more information on logical type, see subsection 4.3.4, page 96.

The CF90 compiler supports the following formats for declaring objects of this type:

```
LOGICAL [ ( [ KIND = ] kind_param ) ] [ , attribute_list :: ] entity_list
```

```
LOGICAL * length_value [ , attribute_list :: ] entity_list
```

For *kind_param* values, see subsection 4.3.4, page 96. The *length_value* values correspond to the *kind_param* values in the following manner:

- On Cray PVP systems and CRAY T3E systems, the values are as follows:

kind_param	*length_value*
1	1
2	2
4	4
8 (default)	8 (default)

Section 5

- On SPARC systems, the values are as follows:

kind_param	*length_value*
1	1
2	2
4 (default)	4 (default)
8	8

The following examples show entity-oriented declaration statements:

```
LOGICAL, ALLOCATABLE :: MASK_1(:), MASK_2(:)
LOGICAL(KIND = WORD), SAVE :: INDICATOR, STATUS
```

The following examples show attribute-oriented declaration statements:

```
LOGICAL MASK_1(:), MASK_2(:)
LOGICAL (KIND = WORD) INDICATOR, STATUS
ALLOCATABLE MASK_1, MASK_2
SAVE INDICATOR, STATUS
```

ANSI/ISO: The Fortran 90 standard does not specify the `LOGICAL *` *length_value* syntax. The CF90 compiler supports this syntax to be compatible with the CF77 compiling system and other FORTRAN 77 compilers. It is recommended that this syntax not be used in any new code.

Character
5.1.6

A `CHARACTER` statement declares the names of entities to be of type character . For more information on character type, see subsection 4.3.5, page 98.

The following is a format for declaring objects of this type:

```
CHARACTER [ char_selector ] [ , attribute_list :: ] entity_list
```

Section 5

The components of this format are defined as follows:

R506	*char_selector*	**is**	*length_selector*
		or	(LEN = *type_param_value* , KIND = *kind_value*)
		or	(*type_param_value* , [KIND =] *kind_value*)
		or	(KIND = *kind_value* [, LEN = *type_param_value*])
R507	*length_selector*	**is**	([LEN =] *type_param_value*)
		or	* *char_length* [,]
R508	*char_length*	**is**	(*type_param_value*)
		or	*scalar_int_literal_constant*
R509	*type_param_value*	**is**	*specification_expr*
		or	*

The optional comma in a *length_selector* is permitted only if no double colon separator appears in the type declaration statement.

A character type declaration can specify a character length that is a nonconstant expression if it appears in a procedure or a procedure interface if it is not a component declaration in a derived-type definition. The length is determined on entry into the procedure and is not affected by any changes in the values of variables in the expression during the execution of the procedure. A character object declared this way that is not a dummy argument is called an *automatic data object*.

The length of a named character entity or a character component in a type definition is specified by the character selector in the type specification unless there is a character length in an entity or component declaration; if so, the character length specifies an individual length and overrides the length in the character selector. If a length is not specified in either a character selector or a character length, the length is 1.

If the length parameter has a negative value, the length of the character entity is 0.

If a scalar integer literal constant is used to specify a character length, it must not include a kind parameter. (This could produce an ambiguity when fixed source form is used.)

A length parameter value of * can be used only in the following ways:

- To declare a dummy argument of a procedure, in which case the dummy argument assumes the length of the associated actual argument when the procedure is invoked.
- To declare a named constant, in which case the length is that of the constant value.
- To declare the result variable for an external function. Any scoping unit that invokes the function must declare the function with a length other than *, or it must access such a declaration by host or use association. When the function is invoked, the length of the result is the value specified in the declaration in the program unit referencing the function. Note that an implication of this rule is that a length of * must not appear in an `IMPLICIT` statement.
- To declare a character pointee.

A function name must not be declared with a length of * if the function is an internal or module function, or if it is array-valued, pointer-valued, or recursive.

The length of a character-valued statement function or statement function dummy argument of type character must be an integer constant expression.

The following examples show entity-oriented character type declaration statements:

```
CHARACTER(LEN = 10, KIND = ASCII), SAVE :: GREETING(2)
CHARACTER(10) :: PROMPT = ”PASSWORD?”
CHARACTER(*), INTENT(IN) :: HOME_TEAM, VISITORS
CHARACTER*3, SAVE :: NORMAL_1, LONGER(9)*20, NORMAL_2
CHARACTER :: GRADE = ”A”
```

Section 5

The following examples show attribute-oriented character type declaration statements:

```
CHARACTER(LEN = 10, KIND = ASCII) :: GREETING
CHARACTER(10) :: PROMPT
CHARACTER(*) :: HOME_TEAM, VISITORS
CHARACTER*3 :: NORMAL_1, LONGER*20, NORMAL_2
CHARACTER GRADE
SAVE :: GREETING, NORMAL_1, LONGER, NORMAL_2
DIMENSION GREETING(2), LONGER(9)
INTENT(IN) :: HOME_TEAM, VISITORS
DATA PROMPT / ”PASSWORD?” /, GRADE / ”A” /
```

Derived type
5.1.7

A `TYPE` declaration statement declares the names of entities to be of the specified user-defined type. The type name appears in parentheses following the keyword `TYPE`. For more information on derived types, see subsection 4.4, page 109.

The following is a format for declaring objects of user-defined type:

`TYPE ( `*type_name*` ) [ , `*attribute_list*` :: ] `*entity_list*

The following examples show entity-oriented derived type declaration statements:

```
TYPE(COLOR), DIMENSION(:), ALLOCATABLE :: HUES_OF_RED
TYPE(PERSON), SAVE :: CAR_POOL(3)
TYPE(PARAGRAPH), SAVE :: OVERVIEW, SUBSTANCE, SUMMARY
```

The following examples show attribute-oriented derived type declaration statements:

```
TYPE(COLOR) :: HUES_OF_RED
TYPE(PERSON) :: CAR_POOL(3)

TYPE(PARAGRAPH) :: OVERVIEW, SUBSTANCE, SUMMARY
DIMENSION :: HUES_OF_RED(:)
ALLOCATABLE :: HUES_OF_RED
SAVE :: CAR_POOL, OVERVIEW, SUBSTANCE, SUMMARY
```

An object of derived type (a structure) must not have the `PUBLIC` attribute if its type is private.

A structure constructor must be used to initialize an object of derived type. Each component of the structure constructor must be an initialization expression. For more information on structure constructors, see subsection 4.5, page 122.

> **Note:** Variables declared to be Cray pointers and pointees through the Cray `POINTER` statement cannot be declared as components of derived types.

Cray pointer (CF90 extension)

5.1.8

The Cray `POINTER` statement declares one variable to be a Cray pointer (that is, to have the Cray pointer data type) and another variable to be its pointee; that is, the value of the Cray pointer is the address of the pointee. This statement has the following format:

> `POINTER` (*pointer_name* , *pointee_name* [(*array_spec*)]) ≡
> ≡ [, (*pointer_name* , *pointee_name* [(*array_spec*)])] ...

Section 5

pointer_name Pointer to the corresponding *pointee_name* . *pointer_name* contains the address of *pointee_name* . Only a scalar variable (other than a structure) can be declared type Cray pointer; constants, arrays, statement functions, and external functions cannot.

pointee_name Pointee of corresponding *pointer_name* ; must be a variable name, array declarator, or array name. A character pointee must be a character variable declared `CHARACTER*(*)`. The value of *pointer_name* is used as the address for any reference to *pointee_name* ; therefore, *pointee_name* is not assigned storage. If *pointee_name* is an array declarator, it can be explicit-shape (with either constant or nonconstant bounds) or assumed-size.

array_spec If present, this must be either an *explicit_shape_spec_list* , with either constant or nonconstant bounds) or an *assumed_size_spec* .

Example:

```
POINTER(P,B),(Q,C)
```

This statement declares Cray pointer `P` and its pointee `B`, and Cray pointer `Q` and pointee `C`; the pointer's current value is used as the address of the pointee whenever the pointee is referenced.

An array that is named as a pointee in a Cray `POINTER` statement is a pointee array. Character pointee arrays are not valid. Its array declarator can appear in a separate type or `DIMENSION` statement or in the pointer list itself. In a subprogram, the dimension declarator can contain references to variables in a common block or to dummy arguments. As with nonconstant bound array arguments to subprograms, the size of each dimension is evaluated on entrance to the subprogram, not when the pointee is referenced. For example:

```
POINTER(IX, X(N,0:M))
```

ANSI/ISO: The Fortran 90 standard does not specify the Cray `POINTER` statement or data type.

Variables declared to be pointers and pointees through the Cray `POINTER` statement cannot be declared as components of derived types.

In addition, pointees must not be deferred-shape or assumed-shape arrays. An assumed-size pointee array is not allowed in a main program unit.

5.1.9 Cray character pointer (CF90 extension)

To define a character pointer, use the `CLOC` intrinsic function. (The `LOC` intrinsic function returns only the word address.)

The `FCD` function can also be used to construct a Cray character pointer. `FCD` has two integer arguments:

- On Cray PVP systems and CRAY T3E systems, the first argument is the word address for the first character of the pointee. On SPARC systems, the first argument is the byte address for the first character of the pointee.
- On all systems, the second argument is the character length of the pointee.

On Cray PVP systems and CRAY T3E systems, if the pointee does not begin on a word boundary, the character offset can be added to the `FCD` result.

Note: Intrinsic functions `CLOC` and `FCD` are not part of the Fortran 90 standard.

The size of an FCD and of a Cray character pointer depends on your platform, so Cray character pointers should not be `EQUIVALENCE`d or storage-associated.

Example:

```
CPTR = FCD(IADRS,ILEN) + IOFFSET
```

Cray character pointers are not optimized; code containing them is not vectorized or autotasked.

Section 5

The following operations are the only ones allowed with Cray character pointers, where *cp* is a character pointer and *i* is an integer. Arithmetic is in bytes (characters), not words or bits.

- *cp* + *i*
- *cp* − *i*
- *i* + *cp*
- *cp* = *cp*
- *cp* *relational_operator* *cp*

Implicit typing

5.2

Each variable, named constant, and function has a type and a name. If the type is not declared explicitly, it is assumed from the first letter of the name. This method of determining type is called *implicit typing*. In each scoping unit, there is in effect a mapping of each of the letters `A, B, ..., Z` (and corresponding lowercase letters) to one of the accessible types or to no type. `IMPLICIT` statements in a scoping unit can be used to specify a mapping different from the default mapping. If a new mapping for a letter is not specified in an `IMPLICIT` statement, the default mapping continues to apply for that letter.

An `IMPLICIT NONE` statement specifies that there is no mapping for any letter and thus all variables, named constants, and functions must be declared in type declaration statements. If the host of a scoping unit contains the `IMPLICIT NONE` statement and the scoping unit contains `IMPLICIT` statements for some letters, the other letters retain the null mapping. This is the only situation in which some initial letters specify an implied type and other initial letters require explicit declarations.

A program unit is treated as if it had a host with the mapping shown in Figure 6. That is, each undeclared variable or function whose name begins with any of the letters `I`, `J`, `K`, `L`, `M`, or `N` is of type integer and all others are of type real.

$$\overbrace{\mathtt{A\ B\ C\ D\ E\ F\ G\ H}}^{\text{Real}}\ \overbrace{\mathtt{I\ J\ K\ L\ M\ N}}^{\text{Integer}}\ \overbrace{\mathtt{O\ P\ Q\ R\ S\ T\ U\ V\ W\ X\ Y\ Z}}^{\text{Real}}$$

Figure 6. Default implicit mapping for a program unit

The `IMPLICIT` statement is defined as follows:

R540	*implicit_stmt*	**is**	`IMPLICIT` *implicit_spec_list*
		or	`IMPLICIT NONE`
R541	*implicit_spec*	**is**	*type_spec* (*letter_spec_list*)
R542	*letter_spec*	**is**	*letter* [– *letter*]

If `IMPLICIT NONE` appears, it must precede any `PARAMETER` statements and there must be no other `IMPLICIT` statements in the scoping unit.

If the – *letter* option appears in a letter specification, the second letter must follow the first alphabetically.

The same letter must not appear as a single letter or be included in a range of letters more than once in all of the `IMPLICIT` statements in a scoping unit.

An `IMPLICIT` statement can be used to specify implicit mappings for user-defined types as well as for intrinsic types.

The `IMPLICIT` statement specifies that all variables, named constants, and functions beginning with the indicated letters are assigned the indicated data type (and type parameters) implicitly. For example, consider the following statement:

```
IMPLICIT COMPLEX (A-C, Z)
```

In this statement, all undeclared variables, named constants, and functions beginning with the letters A, B, C, and Z are of type default complex. If this is the only IMPLICIT statement, undeclared variables, named constants, and functions beginning with I–N will still be of type integer; undeclared variables, named constants, and functions beginning with D–H and O–Y will be of type real.

As another example, consider the following statement:

```
IMPLICIT NONE
```

In this statement, there is no implicit typing in the scoping unit. Each variable and named constant local to the scoping unit, and each external function used in the scoping unit, must be declared explicitly in a type statement. This statement is useful for detecting inadvertent misspellings in a program because misspelled names become undeclared rather than implicitly declared.

The following examples show IMPLICIT statements:

```
IMPLICIT INTEGER (A-G), LOGICAL(KIND = WORD) (M)
IMPLICIT CHARACTER*10 (P, Q)
IMPLICIT TYPE(COLOR) (X-Z)
```

The additional complexity that implicit typing causes in determining the scope of an undeclared variable in a nested scope is explained in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

5.3 Array properties

An array object has the DIMENSION attribute. An array specification determines the array's rank, or number of dimensions. The extents of the dimensions may be declared or left unspecified. If they are left unspecified, the array must also have the ALLOCATABLE or POINTER attribute, or it must be a dummy argument.

Array specifications
5.3.1

There are four formats that an *array_spec* (R512) can take:

R512	*array_spec*	**is**	*explicit_shape_spec_list*
		or	*assumed_shape_spec_list*
		or	*deferred_shape_spec_list*
		or	*assumed_size_spec*

The maximum rank of an array is 7. A scalar is considered to have rank 0.

An array with a *deferred_shape_spec_list* must have the `POINTER` or `ALLOCATABLE` attribute.

An array with an *assumed_shape_spec_list* or an *assumed_size_spec* must be a dummy argument.

Explicit-shape arrays
5.3.1.1

An explicit-shape array has bounds specified in each dimension. Each dimension is specified by an *explicit_shape_spec*, which is defined as follows:

R513	*explicit_shape_spec*	**is**	[*lower_bound* **:**] *upper_bound*
R514	*lower_bound*	**is**	*specification_expr*
R515	*upper_bound*	**is**	*specification_expr*

For more information on *specification_expr*s, see subsection 7.2.9.3, page 284.

The number of sets of bounds specified is the number of dimensions (rank) of the array.

If the lower bound is omitted, the default value is 1.

The value of a lower bound or an upper bound can be positive, negative, or 0.

Section 5

The subscript range of the array in a dimension is the set of integer values between and including the lower and upper bounds, provided the upper bound is not less than the lower bound. If the upper bound is less than the lower bound, the range is empty, the extent in that dimension is 0, and the size of the array is 0.

The expression for a bound could involve variables that cause the expression to have different values each time the procedure in which it is declared is executed. If so, the array must be a dummy argument, a function result, or an automatic array, in which case the actual bounds are determined when the procedure is entered. The bounds of such an array are unaffected by any redefinition or undefinition of the specification variables during the execution of the procedure.

The following are examples of entity-oriented explicit-shape array declarations:

```
REAL Q(-10:10, -10:10, 2)  ! in a main program
   . . .

SUBROUTINE EX1(Z, I, J)             ! in a
  REAL, DIMENSION(2:I + 1, J) :: Z  ! subroutine
      . . .
```

The following are examples of attribute-oriented explicit-shape array declarations:

```
REAL Q
DIMENSION Q(-10:10, -10:10, 2)  ! in a main
                                ! program
. . .
SUBROUTINE EX1(Z, I, J)     ! in
   REAL Z                   ! a
   DIMENSION Z(2:I + 1, J)  ! subroutine
      . . .
```

Section 5

5.3.1.2 Assumed-shape arrays

An assumed-shape array is a dummy argument that takes the shape of the actual argument passed to it. An *assumed_shape_spec* has the following format:

R516	*assumed_shape_spec*	**is**	[*lower_bound*] **:**

The *lower_bound* of the assumed-shape array is the specified lower bound, if present; otherwise it is 1.

The rank is equal to the number of colons in the *assumed_shape_spec_list* .

The upper bound is the extent of the corresponding dimension of the associated array plus the lower bound minus 1.

An assumed-shape array must not have the `POINTER` or `ALLOCATABLE` attribute.

The following examples show an entity-oriented, assumed-shape array declaration:

```
REAL, DIMENSION(2:, :) :: X
```

The following examples show an attribute-oriented, assume-shaped array declaration:

```
SUBROUTINE EX2(A, B, X)
   REAL A(:), B(0:), X
   DIMENSION X(2:, :)
   INTENT(IN) A, B
      . . .
```

As another example, assume that `EX2` is called by the following statement:

```
CALL EX2(U, V, W(4:9, 2:6))
```

Dummy argument `X` is an array with bounds (`2:7, 1:5`). The lower bound of the first dimension is 2 because `X` is declared to have a lower bound of 2. The upper bound is 7 because the dummy argument takes its shape from the actual argument `W`.

Section 5

Deferred-shape arrays
5.3.1.3

A deferred-shape array is either an array pointer or an allocatable array. An array pointer is an array that has the POINTER attribute. Its extent in each dimension is determined when the pointer is allocated or when a pointer assignment statement for the pointer is executed. An allocatable array is an array that has the ALLOCATABLE attribute. Its bounds, and thus its shape, are determined when the array is allocated. In both cases a colon specifies the declared bound; that is, the format of a *deferred_shape_spec* is defined as follows:

R517	*deferred_shape_spec*	**is**	:

The rank is equal to the number of colons in the *deferred_shape_spec_list* .

The bounds of an allocatable array are specified in an ALLOCATE statement when the array is allocated.

The lower bound of each dimension of an array pointer is the result of the LBOUND intrinsic function applied to the corresponding dimension of the target. The upper bound of each dimension is the result of the UBOUND intrinsic function applied to the corresponding dimension of the target. This means that if the bounds are determined by allocation of the pointer, you can specify them. If the bounds are determined by pointer assignment, there are two possible interpretations:

- If the pointer target is a named whole array, the bounds are those declared in the array declaration or those specified when the array was allocated.
- If the pointer target is an array section, the lower bound is 1 and the upper bound is the extent in that dimension.

The bounds and shape of an array pointer or allocatable array are unaffected by any subsequent redefinition or undefinition of variables involved in determination of the bounds.

The following examples show entity-oriented, deferred-shape array declarations:

```
REAL, POINTER :: D(:, :), P(:) ! array pointers
REAL, ALLOCATABLE :: E(:)       ! allocatable array
```

The following examples show attribute-oriented, deferred-shaped array declarations:

```
REAL D
DIMENSION D(:, :), P(:), E(:)
POINTER D, P
ALLOCATABLE E
```

Assumed-size arrays
5.3.1.4

An assumed-size array is a dummy argument array whose size is assumed from that of the associated actual argument. Only the size is assumed. The rank, extents, and bounds (except for the upper bound and extent in the last dimension) are determined by the declaration of the dummy array. The rules for argument association between an actual argument and an assumed-size array are as follows:

- They must have the same initial array element.
- Successive array elements are storage associated. For information on storage association, see subsection 5.10, page 189.
- Declarations for the dummy argument determine the rank. They also determine lower bounds for all dimensions and the extents and upper bounds for all dimensions except the last.
- The size of the actual argument determines the size of the dummy argument as explained in this subsection.

The format of an *assumed_size_spec* is defined as follows:

R518	*assumed_size_spec*	**is**	[*explicit_shape_spec_list* ,] [*lower_bound* **:**] *

The rank of an assumed-size array is the number of explicit-shape specifications plus one.

The size of an assumed-size array is determined as follows:

- If the actual argument associated with the assumed-size dummy argument is an array of any type other than character, the size is that of the actual array.
- If the actual argument associated with the assumed-size dummy array is an array element of any type other than character with a subscript order value of v in an array of size x, the size of the dummy argument is x–v+1. For information on array element order, see subsection 6.4.7, page 222.
- If the actual argument is a character array, character array element, or a character array element substring, and if it begins at character storage unit t of an array with c character storage units, the size of the dummy array is `MAX(INT((`*c*`−`*t*`+1)/`*e*`), 0)` where e is the length of an element in the dummy character array.

If r is the rank of the array, the bounds of the first $r-1$ dimensions are those specified by the explicit-shape specification list, if present. The lower bound of the last dimension is the specified lower bound, if present; otherwise it is 1.

The expression for a bound may involve variables that cause the expression to have different values each time the procedure in which it is declared is executed. If so, the bounds are unaffected by any subsequent redefinition or undefinition of such variables involved in the determination of the bounds.

A function result must not be an assumed-size array.

An assumed-size array must not appear in a context where the shape of the array is required, such as a whole array reference.

The following examples show entity-oriented, assumed-size array declarations:

```
SUBROUTINE EX3(N, S, Y)
   REAL, DIMENSION(N, *) :: S
   REAL Y(10, 5, *)
      . . .
```

The following examples show attribute-oriented, assumed size array declarations:

```
SUBROUTINE EX3(N, S, Y)
   REAL S, Y
   DIMENSION S(N, *), Y(10, 5, *)
      . . .
```

5.3.2 DIMENSION attribute and statement

The dimensions of an array can be specified by the appearance of a DIMENSION attribute or by the appearance of an array specification following the name of the array in a type declaration statement. Both a DIMENSION attribute and an array specification following the name can appear in a declaration statement. In this case, the array specification following the name overrides the array specification following the DIMENSION attribute. A format for a type declaration statement with a DIMENSION attribute is as follows:

type , DIMENSION (*array_spec*) [, *attribute_list*] :: *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
ALLOCATABLE
INTENT
OPTIONAL
POINTER
PARAMETER
PRIVATE
PUBLIC
SAVE
TARGET
```

The type declaration can also contain an *initialization_expr* (R504). An array specification can also appear following a name in several different kinds of statements to declare an array. They are DIMENSION, ALLOCATABLE, POINTER, TARGET, and COMMON statements.

Section 5

The DIMENSION statement is the statement form of the DIMENSION attribute and is defined as follows:

R525	*dimension_stmt*	**is**	DIMENSION [::] *array_name* (*array_spec*) ≡ ≡ [, *array_name* (*array_spec*)] ...

The DIMENSION statement also confers the DIMENSION attribute. It is subject to the same rules and restrictions as the DIMENSION attribute.

The following examples show entity-oriented declarations:

```
INTEGER, DIMENSION(10), TARGET, SAVE :: INDICES
INTEGER, ALLOCATABLE, TARGET :: LG(:, :, :)
```

The following examples show attribute-oriented declarations:

```
INTEGER INDICES, LG(:, :, :)
DIMENSION INDICES(10)
TARGET INDICES, LG
ALLOCATABLE LG
SAVE INDICES
```

The following examples show the array specification in other statements:

```
INTEGER INDICES, LG
TARGET INDICES(10), LG
ALLOCATABLE LG(:, :, :)
SAVE INDICES
```

The following example shows the array specification in a COMMON statement:

```
COMMON /UNIVERSAL/ TIME(80), SPACE(20, 20, 20, 20)
```

5.3.3 ALLOCATABLE attribute and statement

Arrays are the only objects that can have the ALLOCATABLE attribute. An allocatable array is one for which the bounds are determined when an ALLOCATE statement is executed for the array. These arrays must be deferred-shape arrays. The following is a format for a type declaration statement with an ALLOCATABLE attribute:

Section 5

type , ALLOCATABLE [, *attribute_list*] :: *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
DIMENSION (with deferred shape)
PRIVATE
PUBLIC
SAVE
TARGET
```

The format of the ALLOCATABLE statement is defined as follows:

R526	*allocatable_stmt*	**is**	ALLOCATABLE [::] *array_name* ≡ ≡ [(*deferred_shape_spec_list*)] ≡ ≡ [, *array_name* ≡ ≡ [(*deferred_shape_spec_list*)]] ...

The array must not be a dummy argument or function result.

If the array is given the DIMENSION attribute elsewhere, the bounds must be specified as colons (deferred shape).

The ALLOCATABLE statement also confers the ALLOCATABLE attribute. It is subject to the same rules and restrictions as the ALLOCATABLE attribute.

The following examples show entity-oriented declarations:

```
REAL, ALLOCATABLE :: A(:, :)
LOGICAL, ALLOCATABLE, DIMENSION(:) :: MASK1
```

The following examples show attribute-oriented declarations:

```
REAL A(:, :)
LOGICAL MASK1
DIMENSION MASK1(:)
ALLOCATABLE A, MASK1
```

POINTER properties

5.4

Most attributes, when applied to an object, add characteristics that the object would not have otherwise. The POINTER attribute, in some sense, removes a characteristic that an object has. Ordinarily, an object has storage space set aside. If the object has the POINTER attribute, it has no space initially and must not be referenced until space is associated with it. An ALLOCATE statement creates new space for a pointer object. A pointer assignment statement permits the pointer to borrow the space from another object. The space that becomes associated with a pointer is called the pointer 's *target*. The target can change during the execution of a program. A pointer target is either an object or part of an object declared to have the TARGET attribute; or it is an object or part of an object that was created by the allocation of a pointer. A pointer can be assigned the target (or part of the target) of another pointer.

Another way of thinking about a pointer is as a *descriptor* that contains information about the type, type parameters, rank, extents, and location of the pointer's target. Thus, a pointer to a scalar object of type real is different from a pointer to an array of user-defined type.

POINTER attribute and statement

5.4.1

The following is a format for a type declaration statement with a POINTER attribute:

```
type , POINTER [ , attribute_list ] :: entity_decl_list
```

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

DIMENSION (with deferred shape)
OPTIONAL
PRIVATE
PUBLIC
SAVE

Section 5

The POINTER statement also provides a means for declaring pointers. Its format is defined as follows:

R527	*pointer_stmt*	**is**	POINTER [::] *object_name* ≡ ≡ [(*deferred_shape_spec_list*)] ≡ ≡ [, *object_name* ≡ ≡ [(*deferred_shape_spec_list*)]] ...

The target of a pointer can be a scalar or an array.

An array pointer must be declared as a deferred-shape array.

A pointer must not be referenced or defined unless it is associated with a target that can be referenced or defined. (A pointer on the right-hand side of a pointer assignment is not considered a pointer reference.)

The POINTER statement also confers the POINTER attribute. It is subject to the same rules and restrictions as the POINTER attribute.

The following example shows an entity-oriented declaration:

```
TYPE(NODE), POINTER :: CURRENT
REAL, POINTER :: X(:, :), Y(:)
```

The following example shows an attribute-oriented declaration:

```
TYPE(NODE) CURRENT
REAL X(:, :), Y(:)
POINTER CURRENT, X, Y
```

TARGET attribute and statement
5.4.2

Only an object with the TARGET attribute can become the target of a pointer during execution of a program. If an object does not have the TARGET attribute or has not been allocated, no part of it can be accessed through a pointer . The following is a format for a type declaration statement with a TARGET attribute:

type , TARGET [, *attribute_list*] :: *entity_list*

Section 5

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
ALLOCATABLE
DIMENSION
INTENT
OPTIONAL
PRIVATE
PUBLIC
SAVE
```

The type declaration statement can also contain an *initialization_expr* (R504).

The `TARGET` statement also provides a means for specifying pointer targets. It has the following format:

R528	*target_stmt*	**is**	`TARGET` [`::`] *object_name* [`(` *array_spec* `)`] ≡ ≡ [`,` *object_name* [`(` *array_spec* `)`]] ...

The `TARGET` statement also confers the `TARGET` attribute. It is subject to the same rules and restrictions as the `TARGET` attribute.

The following examples show entity-oriented declarations:

```
TYPE(NODE), TARGET :: HEAD_OF_LIST
REAL, TARGET, DIMENSION(100, 100) :: V, W(100)
```

The following examples show attribute-oriented declarations:

```
TYPE(NODE) HEAD_OF_LIST
REAL V, W(100)
DIMENSION V(100, 100)
TARGET HEAD_OF_LIST, V, W
```

5.5 Data initialization and the DATA statement

An entity can be initialized in a type declaration statement. When an initialization expression appears in a declaration for an object that does not have the PARAMETER attribute, the object (which is a variable) is assigned the specified initial value. The same rules apply to the assignment of the initial value as apply when an assignment statement is executed. For example, if the variable is of type real but the value is an integer value, the variable will be assigned the real equivalent of the integer value. If the kind of the variable is different from the kind of the value, the value will be converted to the kind of the variable. Array constructors and broadcast values can be used to initialize arrays, and structure constructors can be used to initialize variables of user-defined type. The format of a type declaration statement that provides an initial value for a variable is as follows:

```
type [ , attribute_list ] :: object_name [ ( array_spec ) ]
  [ * char_length ] = initialization_expr
```

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
DIMENSION
PRIVATE
PUBLIC
SAVE
TARGET
```

For example: `INTEGER :: I = 0`

The PARAMETER attribute can also appear in the *attribute_list*, but in this case, the object is declared to be a named constant.

The value associated with the name cannot be changed during the execution of the program. For example, PI or E can be associated with the familiar mathematical constants to provide more convenient access to these values. Named constants are also used to assign names to values (such as a sales tax rate) that could change at some later time. When a change is necessary, it can be made at one place in the program rather than every place where the value is used. The program can be recompiled to effect the change.

An array name that appears in a declaration statement that contains an *initialization_expr* must have its dimensionality declared in the same statement or a previous statement.

Initialization of a variable in a type declaration statement or any part of a variable in a `DATA` statement implies that the variable has the `SAVE` attribute unless the variable is in a named common block. The automatically acquired `SAVE` attribute may be reaffirmed by the appearance of `SAVE` as an attribute in its type declaration statement or by inclusion of the variable name in a separate `SAVE` statement.

The `DATA` statement is defined as follows:

R529	*data_stmt*	**is**	`DATA` *data_stmt_set* ≡
			≡ [[,] *data_stmt_set*] ...
R530	*data_stmt_set*	**is**	*data_stmt_object_list* ≡
			≡ / *data_stmt_value_list* / ≡
			≡ [[,] *data_stmt_object_list* ≡
			≡ / *data_stmt_value_list* /] ...
R531	*data_stmt_object*	**is**	*variable*
		or	*data_implied_do*
R532	*data_stmt_value*	**is**	[*data_stmt_repeat* *] ≡
			≡ *data_stmt_constant*
R533	*data_stmt_constant*	**is**	*scalar_constant*
		or	*signed_int_literal_constant*
		or	*signed_real_literal_constant*
		or	*structure_constructor*
		or	*boz_literal_constant*
CRI		**or**	*typeless_constant*
R534	*data_stmt_repeat*	**is**	*scalar_int_constant*

Section 5

R535	*data_implied_do*	**is**	(*data_i_do_object_list* , ≡ ≡ *data_i_do_variable* = ≡ ≡ *scalar_int_expr* , ≡ ≡ *scalar_int_expr* ≡ ≡ [, *scalar_int_expr*])
R536	*data_i_do_object*	**is**	*array_element*
		or	*scalar_structure_component*
		or	*data_implied_do*
R537	*data_i_do_variable*	**is**	*scalar_int_variable*
CRI	*typeless_constant*	**is**	*octal_typeless_constant*
		or	*hexadecimal_typeless_constant*
		or	*binary_typeless_constant*
CRI	*octal_typeless_constant*	**is**	*digit* [*digit* ...] B
		or	O" *digit* [*digit* ...] "
		or	O' *digit* [*digit* ...] '
		or	" *digit* [*digit* ...] "O
		or	' *digit* [*digit* ...] 'O
CRI	*hexadecimal_typeless_constant*	**is**	X' *hex_digit* [*hex_digit* ...] '
		or	X" *hex_digit* [*hex_digit* ...] "
		or	' *hex_digit* [*hex_digit* ...] 'X
		or	" *hex_digit* [*hex_digit* ...] "X
		or	Z' *hex_digit* [*hex_digit* ...] '
		or	Z" *hex_digit* [*hex_digit* ...] "
CRI	*binary_typeless_constant*	**is**	B' *bin_digit* [*bin_digit* ...] '
		or	B" *bin_digit* [*bin_digit* ...] "

The following notes pertain to the preceding format:

- *digit* must have one of the values 0 through 7 in *octal_typeless_constant*
- *digit* must have a value of 0 or 1 in *binary_typeless_constant*
- The B, O, X, and Z characters can be in uppercase or lowercase.

Note that a constant value cannot be an array constructor. An array can be initialized by using the array name as the *data_stmt_object* and supplying values for all the elements of the array using a *data_implied_do* .

> **ANSI/ISO:** The Fortran 90 standard does not specify the *typeless_constant* .

If an object is of type character or logical, the constant used for initialization must be of the same type.

If an object is of type real or complex, the corresponding constant must be of type integer, real, or complex. The CF90 compiler permits a single-precision real object to be initialized with a BOZ, typeless, or character (used as Hollerith) constant. No conversion of the BOZ value, typeless value, or character constant takes place. If the object is of type integer, the corresponding constant must be of type integer, real, or complex; or, if the initialization is specified in a `DATA` statement, the corresponding constant can be a BOZ constant.

The CF90 compiler permits an integer object to be initialized with a BOZ, typeless, or character (used as Hollerith) constant in a type declaration statement. The CF90 compiler also allows an integer object to be initialized with a typeless or character (used as Hollerith) constant in a `DATA` statement.

> **ANSI/ISO:** The Fortran 90 standard does not specify typeless or character (used as Hollerith) constants in initializations. Neither does it allow BOZ constants to be used in type declaration statement initializations.

If an object is of derived type, the corresponding structure constructor must be of the same type.

The value of the constant, structure constructor (in a `DATA` statement), or initialization expression (in a type declaration statement) must be such that its value could be assigned to the corresponding variable using an intrinsic assignment statement. The variable becomes initially defined with the value of the constant.

A variable, or the same part of a variable, must not be initialized more than once in an executable program.

The following items cannot be initialized:

- A dummy argument
- An object made accessible by use or host association
- A function result
- An automatic object
- A pointer
- An allocatable array
- An external or intrinsic procedure

> **ANSI/ISO:** The Fortran 90 standard does not allow initialization of objects in named common blocks except from within a `BLOCKDATA` program unit. The Fortran 90 standard does not allow initialization of objects in a blank common block.

For an object being initialized, any subscript, section subscript, substring starting point, or substring ending point must be an integer initialization expression.

Each component of a structure constructor used for initialization must be an initialization expression.

A variable that appears in a `DATA` statement and is thereby declared and typed implicitly can appear in a subsequent type declaration statement only if that declaration confirms the implicit declaration. An array name, array section, or array element appearing in a `DATA` statement must have had its array properties established previously.

An array element or structure component that appears in a `DATA` statement must not have a constant parent.

The `DATA` statement repeat factor value must be positive or zero. If it is a named constant, the value must have been specified in a prior statement in the scoping unit that contains the `DATA` statement or must have been made accessible by use or host association.

A subscript in an array element of an implied-`DO` list must contain as operands only constants or `DO` variables of the containing implied-`DO`s.

The scalar integer loop control expressions in an implied-DO must contain as operands only constants or DO variables of the containing implied-DOs. Each operation must be an intrinsic operation.

The data object list is expanded to form a sequence of scalar variables. An array or array section is equivalent to the sequence of its array elements in array element order. A *data_implied_do* is expanded to form a sequence of array elements and structure components, under the control of the implied- DO variable, as in the DO construct. A zero-sized array or an implied- DO with an iteration count of 0 contributes no variables to the expanded list, but a character variable declared to have zero length does contribute a variable to the list.

The data value list is expanded to form a sequence of scalar values. Each value must be a constant or constant expression (structure constructor). If a value is represented by a named constant, the named constant must be specified prior to the DATA statement. A DATA statement repeat factor indicates the number of times the following constant value is to be included in the sequence. If the repeat factor is 0, the following value is ignored.

Scalar variables and values of the expanded sequence must be in one-to-one correspondence. Each value specifies the initial value for the corresponding variable. The lengths of the two expanded sequences must be the same.

> **ANSI/ISO:** If the last item in the *data_object_list* is an array name, the value list can contain fewer values than the number of elements in the array . Any element that is not assigned a value is undefined.

The following examples show type declaration statement initializations:

```
CHARACTER(LEN = 10) :: NAME = "JOHN DOE"
INTEGER, DIMENSION(0:9) :: METERS = (/ (0, I = 1, 10) /)
TYPE(PERSON) :: ME = PERSON(21, "JOHN SMITH"), &
   YOU = PERSON(35, "FRED BROWN")
INTEGER :: BIRD(3) = 1
REAL :: SKEW(100,100) = RESHAPE ( (/((1.0, K = 1, J-1), &
       (0.0, K = J, 100), J = 1, 100)/), (/ 100, 100 /) )
```

The following are examples of DATA statement initializations:

```
CHARACTER*10 NAME
INTEGER METERS(0:9)
DATA NAME /"JOHN DOE"/, METERS /10*0/

TYPE(PERSON) ME, YOU
DATA ME /PERSON(21, "JOHN SMITH")/
DATA YOU%AGE, YOU%NAME /35, "FRED BROWN"/
INTEGER BIRD(3)
DATA BIRD /3*1/
REAL SKEW(100, 100)
DATA ((SKEW (K, J), K = 1, J-1), J = 1, 100) /4950 * 1.0/
DATA ((SKEW (K, J), K = J, 100), J = 1, 100) /5050 * 0.0/
```

In both forms, the character variable NAME is initialized with the value JOHN DOE with padding on the right because the length of the constant is less than the length of the variable. All ten elements of the integer array METERS are initialized to 0; an array constructor is used in the type declaration statement form; a repeat factor is used for the DATA statement form. ME and YOU are structures declared using the user-defined type PERSON defined in subsection 4.6, page 125. In both forms ME is initialized using a structure constructor. In the DATA statement form YOU is initialized by supplying a separate value for each component.

In the type declaration statement form, the value 1 is broadcast to all 3 elements of BIRD. In the DATA statement form, a value must be supplied for each element of BIRD.

In both forms, the two-dimensional array SKEW is initialized so that the lower triangle is 0 and the strict upper triangle is 1. The RESHAPE intrinsic function is required in the first form because SKEW is of rank 2. Repeat factors are used in the second form.

Section 5

PARAMETER attribute and statement
5.5.1

A constant can be given a name in a type declaration statement with the PARAMETER attribute or in a PARAMETER statement. The following is a format for a type declaration statement with a PARAMETER attribute:

type , PARAMETER [, *attribute_list*] :: *name* = *initialization_expression*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
DIMENSION
PRIVATE
PUBLIC
```

The *initialization_expression* must be present.

More than one named constant can be specified in a single type declaration statement; see the examples in this subsection.

The named constant becomes defined with the value determined from the initialization expression in accordance with the rules for intrinsic assignment. Any named constant that appears in the initialization expression must meet one of the following conditions:

- Be defined previously in this type declaration statement or in a previous type declaration statement
- Be accessible through host or use association

If the named constant is an array, it must have its array properties declared in this statement or in a previous statement in the same scoping unit.

The PARAMETER statement also provides a means of defining a named constant. Its format is defined as follows:

R538	*parameter_stmt*	**is**	PARAMETER (*named_constant_def_list*)
R539	*named_constant_def*	**is**	*named_constant* = *initialization_expr*

The PARAMETER statement also confers the PARAMETER attribute. It is subject to the same rules and restrictions as the PARAMETER attribute.

The PARAMETER attribute must not be specified for dummy arguments, functions, or objects in a common block.

A named constant that appears in a PARAMETER statement and is thereby declared and typed implicitly may appear in a subsequent type declaration statement only if that declaration confirms the implicit declaration.

A named array constant appearing in a PARAMETER statement must have had its array properties established previously.

A named constant must not appear in a format specification because of a possible ambiguity.

The following examples show entity-oriented declarations:

```
INTEGER, PARAMETER :: STATES = 50
INTEGER, PARAMETER :: M = MOD(28, 3),  &
       NUMBER_OF_SENATORS = 2 * STATES
```

The following examples show attribute-oriented declarations:

```
INTEGER STATES, M, NUMBER_OF_SENATORS
PARAMETER(STATES = 50)
PARAMETER(M = MOD(28, 3),  &
      NUMBER_OF_SENATORS = 2 * STATES)
```

Object accessibility and use

5.6

Several attributes indicate where an object can be accessed and how it can be used. Some of these attributes apply only to objects in a module and others apply only to dummy arguments or other variables that are declared in a subprogram.

PUBLIC and PRIVATE attributes and statements

5.6.1

The PUBLIC and PRIVATE attributes control access to type definitions, variables, functions, and named constants in a module. The PUBLIC attribute declares that entities in a module are available outside the module by use association; the PRIVATE attribute prevents access outside the module by use association. The default accessibility is PUBLIC, but it can be changed to PRIVATE.

The following formats are for type declaration statements with `PUBLIC` and `PRIVATE` attributes:

type `, PUBLIC [ ,` *attribute_list* `]::` *entity_decl_list*

type `, PRIVATE [ ,` *attribute_list* `]::` *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
ALLOCATABLE
DIMENSION
EXTERNAL
INTRINSIC
PARAMETER
POINTER
SAVE
TARGET
```

The type declaration statement can also contain an *initialization_expr* (R504).

`PUBLIC` and `PRIVATE` specifications can also appear in the derived-type statement of a derived-type definition in a module to specify the accessibility of the type definition, as shown in the following:

`TYPE , PUBLIC ::` *type_name*

`TYPE , PRIVATE ::` *type_name*

If a `PRIVATE` statement appears inside a type definition, it specifies that, although the type may be accessible outside the module, its components are private.

For more information on derived-type definitions, see subsection 4.4.1, page 110.

PUBLIC and PRIVATE statements provide another means for controlling the accessibility of variables, functions, type definitions, and named constants. PUBLIC and PRIVATE statements can control the accessibility of some entities that do not have a type; these are subroutines, generic specifiers, and namelist groups. The formats for PUBLIC and PRIVATE statements are defined as follows:

R521	*access_stmt*	**is**	*access_spec* [[::] *access_id_list*]
R510	*access_spec*	**is**	PUBLIC
		or	PRIVATE
R522	*access_id*	**is**	*use_name*
		or	*generic_spec*

Specify one of the following for *generic_spec* (R1206):

generic_name
OPERATOR (*defined_operator*)
ASSIGNMENT (=)

PUBLIC and PRIVATE statements can appear only in a module.

The PUBLIC and PRIVATE statements also confer the PUBLIC or PRIVATE attribute. They are subject to the same rules and restrictions as the PUBLIC and PRIVATE attributes.

A *use_name* can be the name of a variable, procedure, derived type, named constant, or namelist group.

Generic specifications are explained further in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903. The following are examples of `PUBLIC` and `PRIVATE` statements that might be used with generic specifications:

```
PUBLIC HYPERBOLIC_COS, HYPERBOLIC_SIN      ! generic names
PRIVATE MY_COS_RAT, MY_SIN_RAT             ! specific names
PRIVATE MY_COS_INF_PREC, MY_SIN_INF_PREC   ! specific names
PUBLIC :: OPERATOR ( .MYOP. ), OPERATOR (+), ASSIGNMENT (=)
```

Only one `PUBLIC` or `PRIVATE` statement with an omitted *access_id* list is permitted in the scoping unit of a module. It determines the default accessibility of the module.

The default accessibility of entities defined in a module is `PUBLIC`. A `PUBLIC` statement without an *access_id* list can appear in the module to confirm the default accessibility. A `PRIVATE` statement without an *access_id* list can appear in the module to change the default accessibility.

A procedure that has a generic identifier that is public is accessible through the generic identifier even if its specific name is private. The converse is also true. That is, a module procedure that is public, but whose generic identifier is private, is still accessible through its specific name.

A module procedure that has an argument of a private type or a function result of a private type must be private and must not have a generic identifier that is public.

The following examples show entity-oriented declarations:

```
REAL, PUBLIC :: GLOBAL_X
TYPE, PRIVATE :: LOCAL_DATA
   LOGICAL :: FLAG
   REAL, DIMENSION(100) :: DENSITY
END TYPE LOCAL_DATA
```

The following examples show attribute-oriented declarations:

```
REAL GLOBAL_X
PUBLIC GLOBAL_X
TYPE LOCAL_DATA
   LOGICAL FLAG
   REAL DENSITY
   DIMENSION DENSITY(100)
END TYPE LOCAL_DATA
PRIVATE LOCAL_DATA
```

The following example shows a public type declaration with private components:

```
TYPE LIST_ELEMENT
   PRIVATE
   REAL VALUE
   TYPE(LIST_ELEMENT), POINTER :: NEXT, FORMER
END TYPE LIST_ELEMENT
```

The following example shows how to override the default accessibility:

```
MODULE M
   PRIVATE
   REAL R, K, TEMP(100)           ! R, K, and TEMP are private
   REAL, PUBLIC :: A(100), B(100) ! A and B are public
END MODULE M
```

5.6.2 INTENT attribute and statement

The INTENT attribute specifies the intended use of a dummy argument. If specified, it can help detect errors, provide information for readers of the program, and give the compiler information that can be used to make the code more efficient. It is particularly valuable in creating software libraries.

Some dummy arguments are referenced but not redefined within the subprogram; some are defined before being referenced within the subprogram; others can be referenced before being redefined. INTENT has three forms: IN, OUT, and INOUT, which correspond respectively to the preceding three situations.

If the intent of an argument is `IN`, the subprogram must neither change the value of the argument nor must the argument become undefined during the course of the subprogram. If the intent is `OUT`, the subprogram must not use the argument before it is defined, and it must be definable. If the intent is `INOUT`, the argument can be used to communicate information to the subprogram and return information; it must be defined on entry into the subprogram and must be definable. If no intent is specified, the use of the argument is subject to the limitations of the associated actual argument. For example, the actual argument may be a constant (for example, `2`) or a more complicated expression (for example, `N+2`), and in these cases the dummy argument can be referenced but not defined.

The following is a format for a type declaration statement with an `INTENT` attribute:

type `,` `INTENT` `(` *intent_spec* `)` `[` `,` *attribute_list* `]` `::` *decl_list*

For *intent_spec* (R511), specify one of the following arguments:

```
IN
OUT
INOUT
```

The *attribute_list* can contain one of the following attributes:

```
DIMENSION
OPTIONAL
TARGET
```

The `INTENT` statement also provides a means of specifying an intent for an argument. Its format is defined as follows:

R519	*intent_stmt*	**is**	`INTENT (` *intent_spec* `) [ :: ]` ≡ ≡ *dummy_arg_name_list*
R511	*intent_spec*	**is**	`IN`
		or	`OUT`
		or	`INOUT`

Section 5

The INTENT attribute can be specified only for dummy arguments.

An INTENT statement can appear only in the specification part of a subprogram or interface body.

An intent must not be specified for a dummy argument that is a dummy procedure because it is not possible to change the definition of a procedure. Intent for a dummy pointer must not be specified either.

The INTENT statement also confers the INTENT attribute. It is subject to the same rules and restrictions as the INTENT attribute.

The following examples show entity-oriented declarations:

```
SUBROUTINE MOVE(FROM, TO)
   USE PERSON_MODULE
   TYPE(PERSON), INTENT(IN) :: FROM
   TYPE(PERSON), INTENT(OUT) :: TO
      . . .
SUBROUTINE SUB(X, Y)
   INTEGER, INTENT(INOUT) :: X, Y
      . . .
```

The following examples show attribute-oriented declarations:

```
SUBROUTINE MOVE(FROM, TO)
   USE PERSON_MODULE
   TYPE(PERSON) FROM, TO
   INTENT(IN) FROM
   INTENT(OUT) TO
      . . .
SUBROUTINE SUB(X, Y)
   INTEGER X, Y
   INTENT(INOUT) X, Y
      . . .
```

OPTIONAL attribute and statement

5.6.3

The OPTIONAL attribute allows a procedure reference to omit arguments with this attribute. The PRESENT intrinsic function can be used to test the presence of an optional argument in a particular invocation and this test can be used to control the subsequent processing in the procedure. If the argument is not present, the subprogram can supply a default value or it can use an algorithm that is not based on the presence of the argument.

The following is a format for a type declaration statement with an OPTIONAL attribute:

type , OPTIONAL [, *attribute_list*] :: *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
DIMENSION
EXTERNAL
INTENT
POINTER
TARGET
```

The OPTIONAL statement also provides a means for specifying an argument that can be omitted. Its format is defined as follows:

R520	*optional_stmt*	**is**	OPTIONAL [::] *dummy_arg_name_list*

The OPTIONAL attribute can be specified only for dummy arguments.

An OPTIONAL statement can appear only in the scoping unit of a subprogram or interface body.

The OPTIONAL statement also confers the OPTIONAL attribute. It is subject to the same rules and restrictions as the OPTIONAL attribute.

The following examples show entity-oriented declarations in a program fragment:

```
CALL SORT_X(X = VECTOR_A)
   . . .
SUBROUTINE SORT_X(X, SIZEX, FAST)
   REAL, INTENT(INOUT) :: X (:)
   INTEGER, INTENT(IN), OPTIONAL :: SIZEX
   LOGICAL, INTENT(IN), OPTIONAL :: FAST
      . . .

   INTEGER TSIZE

      . . .
   IF (PRESENT(SIZEX)) THEN
      TSIZE = SIZEX
   ELSE
      TSIZE = SIZE(X)
   END IF

   IF (.NOT. PRESENT(FAST) .AND. TSIZE > 1000) THEN
      CALL QUICK_SORT(X)
   ELSE
      CALL BUBBLE_SORT(X)
   END IF
      . . .
```

The following examples show attribute-oriented declarations to be inserted in the same program fragment:

```
SUBROUTINE SORT_X(X, SIZEX, FAST)
   REAL X(:)
   INTENT(INOUT) X
   INTEGER SIZEX
   LOGICAL FAST
   INTENT(IN) SIZEX, FAST
   OPTIONAL SIZEX, FAST
      . . .
   INTEGER TSIZE
      . . .
```

`SAVE` attribute and statement

5.6.4

Variables with the `SAVE` attribute retain their value and their definition, association, and allocation status after the subprogram in which they are declared completes execution. Variables without the `SAVE` attribute cannot be depended on to retain their value and status, although the CF90 compiler treats named common blocks as if they had the `SAVE` attribute. The `SAVE` attribute should always be specified for an object or the object's common named block, if it is necessary for the object to retain its value and status.

Objects declared in a module can be given the `SAVE` attribute, in which case they always retain their value and status when a procedure that uses the module completes execution. Objects in modules must be in continual use in order to retain their values.

Objects declared in recursive subprograms can be given the `SAVE` attribute. Such objects are shared by all instances of the subprogram.

Any object that is data initialized (in a `DATA` statement or a type declaration statement) has the `SAVE` attribute by default.

The following is a format for a type declaration statement with a `SAVE` attribute:

type `, SAVE [ ,` *attribute_list* `] ::` *entity_decl_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
ALLOCATABLE
DIMENSION
POINTER
PRIVATE
PUBLIC
TARGET
```

The type declaration statement can also contain an *initialization_expr* (R504).

The `SAVE` statement provides a means for specifying the `SAVE` attribute for objects and also for named common blocks. Its format is defined as follows:

R523	*save_stmt*	**is**	`SAVE` [[`::`] *saved_entity_list*]
R524	*saved_entity*	**is**	*data_object_name*
		or	/ *common_block_name* /

A `SAVE` statement without a saved entity list is treated as though it contained the names of all items that could be saved in the scoping unit. The CF90 compiler allows you to insert multiple `SAVE` statements without entity lists in a scoping unit.

> **ANSI/ISO:** The Fortran 90 standard permits only one `SAVE` statement without an entity list in a scoping unit.

If `SAVE` appears in a main program as an attribute or a statement, it has no effect.

The following objects must not be saved:

- A procedure
- A function result
- A dummy argument
- An automatic data object
- An object in a common block
- A namelist group

A variable in a common block cannot be saved individually; the entire named common block must be saved if you want any variables in it to be saved.

A named common block saved in one scoping unit of a program is saved throughout the program.

> **ANSI/ISO:** The Fortran 90 standard states that if a named common block is saved in one scoping unit of a program, it must be saved in every scoping unit of the program in which it is defined (other than the main program).

If a named common block is specified in a main program, it is available to any scoping unit of the program that specifies the named common block; it does not need to be saved.

The `SAVE` statement also confers the `SAVE` attribute. It is subject to the same rules and restrictions as the `SAVE` attribute.

The following example shows an entity-oriented declaration:

```
CHARACTER(LEN = 12), SAVE :: NAME
```

The following example shows an attribute-oriented declaration:

```
CHARACTER*12 NAME
SAVE NAME
```

The following example shows saving objects and named common blocks:

```
SAVE A, B, /BLOCKA/, C, /BLOCKB/
```

Procedure properties

5.7

If an external or dummy procedure is to be an actual argument to a subprogram, the procedure name must be declared `EXTERNAL`. (A *dummy procedure* is a dummy argument that is a procedure.) If an external procedure has the same name as an intrinsic procedure, again the name must be declared `EXTERNAL`. When this occurs, the intrinsic procedure of that name is no longer accessible to that program unit. If an intrinsic procedure is to be an actual argument, the name of the procedure must be declared `INTRINSIC`. The *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903, discusses further the usage of these attributes.

Because only functions, not subroutines, are declared to have a type (the type of the result), only function names can appear in type declaration statements. The `EXTERNAL` and `INTRINSIC` attributes in type declaration statements therefore apply only to functions. The `EXTERNAL` and `INTRINSIC` statements can be used to specify properties of subroutines, and the `EXTERNAL` statement can specify block data program units. For information on block data program units, see the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

5.7.1 `EXTERNAL` attribute and statement

The EXTERNAL attribute in a type declaration statement indicates that a name is the name of an external function or a dummy function and permits the name to be used as an actual argument.

The following is a format for a type declaration statement with an EXTERNAL attribute:

type , EXTERNAL [, *attribute_list*] :: *function_name_list*

Subject to the rules governing combinations of these attributes, *attribute_list* can contain the following:

```
OPTIONAL
PRIVATE
PUBLIC
```

An interface block can be used to describe the interface of an external function. A function described by an interface block has the EXTERNAL attribute by default, so the function name cannot also be given the EXTERNAL attribute by any other means. Note that an interface block specifies the EXTERNAL attribute for all procedures in the interface block, with the exception of module procedures specified in MODULE PROCEDURE statements within the block. For information on interface blocks, see the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

The EXTERNAL statement provides a means for declaring subroutines and block data program units, as well as functions, to be external. Its format is defined as follows:

R1207	*external_stmt*	**is**	EXTERNAL *external_name_list*

Each external name must be the name of an external procedure, a dummy argument, or a block data program unit.

If a dummy argument is specified to be EXTERNAL, the dummy argument is a dummy procedure.

The EXTERNAL statement also confers the EXTERNAL attribute. It is subject to the same rules and restrictions as the EXTERNAL attribute.

The following examples of entity-oriented declarations:

```
SUBROUTINE SUB(FOCUS)
   INTEGER, EXTERNAL :: FOCUS
   LOGICAL, EXTERNAL :: SIN
```

The following example shows an attribute-oriented declaration:

```
SUBROUTINE SUB (FOCUS)
   INTEGER FOCUS
   LOGICAL SIN
   EXTERNAL FOCUS, SIN
```

FOCUS is declared to be a dummy procedure. SIN is declared to be an external procedure. Both are functions. To declare an external subroutine, the EXTERNAL statement or an interface block must be used because a subroutine does not have a type, and thus its attributes cannot be specified in a type declaration statement. The specific and generic name SIN of the intrinsic function SIN is no longer available to subroutine SUB.

5.7.2 **INTRINSIC attribute and statement**

The INTRINSIC attribute in a type declaration statement indicates that a name is the name of an intrinsic function and permits the names of some INTRINSIC functions to be used as actual arguments.

The following is a format for a type declaration statement with an INTRINSIC attribute:

type , INTRINSIC [, *attribute_list*] :: *intrinsic_function_name_list*

For *attribute_list* , specify one of the following attributes:

```
PRIVATE
PUBLIC
```

The `INTRINSIC` statement provides a means for declaring intrinsic subroutines, as well as functions. Its format is defined as follows:

R1208	*intrinsic_stmt*	**is**	`INTRINSIC` *intrinsic_procedure_name_list*

Each *intrinsic_procedure_name* must be the name of an intrinsic procedure.

A name must not be declared to be both `EXTERNAL` and `INTRINSIC` in a scoping unit.

A type can be specified for an intrinsic function even though it has a type as specified in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903. If a type is specified for the generic name of an intrinsic function, it does not remove the generic properties of the function name.

The `INTRINSIC` statement also confers the `INTRINSIC` attribute. It is subject to the same rules and restrictions as the `INTRINSIC` attribute.

The CF90 compiler has implemented intrinsic procedures in addition to the ones required by the standard. These procedures have the status of intrinsic procedures, but programs that use them may not be portable. It is recommended that such procedures be declared `INTRINSIC` to allow other processors to diagnose whether or not they are intrinsic for those processors.

The following is an example of an entity-oriented declaration:

```
REAL, INTRINSIC :: SIN, COS
```

The following is an example of an attribute-oriented declaration:

```
REAL SIN, COS
INTRINSIC SIN, COS
```

Because the interfaces of intrinsic procedures are known to the compiler, it is not necessary to specify a type for them, but it is not incorrect to do so.

Automatic data objects

5.8

Automatic data objects are especially useful as working storage in a procedure. These objects can be declared only in procedures or procedure interfaces; they are created when the procedure is entered and disappear when the procedure completes execution. They can be created the same size as an argument to the procedure, so they can be tailored to each invocation.

There are two kinds of automatic data objects: automatic arrays of any type and objects of type character.

An *automatic data object* is one that depends on the value of a nonconstant expression and is not a dummy argument. Automatic arrays are those whose size depends on a value used in a bound expression.

The following are examples of automatic data objects:

```
SUBROUTINE SUB (N, DUMMY_ARRRAY)
COMMON /CB/ K
INTEGER AUTO_ARRAY(N)       ! Automatic array.
INTEGER(K*2) CH             ! Automatic character variable.
INTEGER DUMMY_ARRAY(K,N)    ! Not an automatic array
                            ! because it is a dummy
                            ! argument, not a local array.
. . .
END SUBROUTINE
```

In Fortran 90, the term *automatic object* does not include noncharacter scalar local variables. For an array, the extents in each dimension are determined when the procedure is entered. For a character object, the length is determined when the procedure is entered. Apart from dummy arguments, this is the only character object whose length can vary. For arrays, extents can vary for allocatable arrays and array pointers as well as dummy arguments. An automatic object is not a dummy argument, but it is declared with a specification expression that is not a constant expression. The specification expression can be the length of the character object or the bounds of the array. Automatic objects cannot be saved or initialized.

In the following example, `C` is an automatic array and `MESSAGE` is an automatic character object:

```
SUBROUTINE SWAP_ARRAYS(A, B, A_NAME, B_NAME)
   REAL, DIMENSION(:), INTENT(INOUT) :: A, B
   CHARACTER(LEN = *), INTENT(IN)      :: A_NAME, B_NAME

   REAL C(SIZE (A))
   CHARACTER (LEN = LEN(A_NAME) + LEN(B_NAME) + 17) MESSAGE

   C = A
   A = B
   B = C

   MESSAGE = A_NAME // " and " // B_NAME // " are swapped"
   PRINT *,  MESSAGE
END SUBROUTINE SWAP_ARRAYS
```

`NAMELIST` statement

5.9

A `NAMELIST` statement establishes the name for a collection of objects that can then be referenced by the group name in certain I/O statements. The `NAMELIST` statement is defined as follows:

R543	*namelist_stmt*	**is**	`NAMELIST` / *namelist_group_name* / ≡ ≡ *namelist_group_object_list* [[,] ≡ ≡ / *namelist_group_name* / ≡ ≡ *namelist_group_object_list*] ...
R544	*namelist_group_object*	**is**	*variable_name*

A variable in the variable name list must not be an array dummy argument with nonconstant bounds, a variable with assumed character length, an automatic object, a pointer, an object of a type that has a pointer component at any level, an allocatable array, or a subobject of any of the preceding objects.

If a namelist group name has the PUBLIC attribute, no item in the namelist group object list can have the PRIVATE attribute.

The order in which the data objects (variables) are specified in the NAMELIST statement determines the order in which the values appear on output.

A namelist group name can occur in more than one NAMELIST statement in a scoping unit. The variable list following each successive appearance of the same namelist group name in a scoping unit is treated as a continuation of the list for that namelist group name.

A variable can be a member of more than one namelist group.

A variable must have its type, type parameters, and shape specified previously in the same scoping unit, or it must be determined by implicit typing rules. If a variable is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm this implicit type. The following is an example of a NAMELIST statement:

```
NAMELIST /N_LIST/ A, B, C
```

Storage association

5.10

Generally , the physical storage units or storage order for data objects cannot be specified. However, the COMMON, EQUIVALENCE, and SEQUENCE statements provide sufficient control over the order and layout of storage units to permit data to share storage units.

The COMMON statement provides a means of sharing data between program units. The EQUIVALENCE statement provides a means whereby two or more objects can share the same storage units.

In Fortran 90, modules, pointers, allocatable arrays, and automatic data objects provide additional tools for sharing data and managing storage. The SEQUENCE statement was introduced in Fortran 90 to define a storage order for structures. This permits structures to appear in common blocks and be equivalenced. The SEQUENCE statement can appear only in derived-type definitions to define sequence types. The components of a sequence type have an order in storage sequences that is the order of their appearance in the type definition.

Storage units
5.10.1

FORTRAN 77 defined two kinds of storage units, numeric and character. Fortran 90 introduces new types (the nondefault types), user-defined types, and pointers. Objects of these types and pointers cannot be accommodated by the two storage units defined in FORTRAN 77. It is not desirable to specify storage units for the space that these objects occupy. Fortran 90 uses the term *unspecified storage unit* for these objects. A new Fortran 90 object (a pointer, an object of nondefault type, or a structure containing components that are of nondefault types or are pointers) is said to occupy an unspecified storage unit, but this unit is different for each different sort of object. A pointer occupies a single unspecified storage unit that is different from that of any nonpointer object and can be different for each combination of type, type parameters, and rank.

There are two kinds of structures, sequence structures and nonsequence structures, depending on whether or not the type definition contains a `SEQUENCE` statement. A nonsequence structure occupies a single unspecified storage unit that is different for each type. The three kinds of sequence structures are as follows:

1. Numeric sequence structures (containing only numeric and logical entities of default kind)
2. Character sequence structures (containing only character entities)
3. Sequence structures (containing a mixture of components including objects that occupy numeric, character, and unspecified storage units)

Table 7 lists objects of various types and attributes and the storage units they occupy.

Table 7. Types, attributes, and storage

Types and attributes of object	Storage units
Default integer	1 numeric
Default real	1 numeric
Logical	1 numeric

Section 5

Table 7. Types, attributes, and storage (continued)

Types and attributes of object	Storage units
Double precision	2 numeric
Default complex	2 numeric
Character of length 1	1 character
Character of length *s*	*s* characters
Nondefault integer	1 unspecified
Real other than default real or double precision	1 unspecified
Nondefault complex	1 unspecified
Nonsequence structure	1 unspecified
Numeric sequence structure	*n* numeric, where *n* is the number of numeric storage units the structure occupies
Character sequence structure	*n* characters, where *n* is the number of character storage units the structure occupies
Sequence structure	The sum of the storage sequences of all the ultimate components of the structure.
Any type with the `POINTER` attribute	1 unspecified
Any intrinsic or sequence type with the `DIMENSION` attribute	The size of the array times the number of storage units for the type (will appear in array element order)

Table 7. Types, attributes, and storage (continued)

Types and attributes of object	Storage units
Any nonintrinsic or nonsequence type with the `DIMENSION` attribute	One unspecified storage unit for each element of the array
Any type with the `POINTER` attribute and the `DIMENSION` attribute	1 unspecified

Storage sequence
5.10.2

A *storage sequence* is an ordered sequence of storage units. The storage units can be elements in an array, characters in a character variable, components in a sequence structure, or variables in a common block. A sequence of storage sequences forms a composite storage sequence. The order of the storage units in such a composite sequence is the order of the units in each constituent taken in succession, ignoring any zero-sized sequences.

Storage is associated when the storage sequences of two different objects have some storage in common. This permits two or more variables to share the same storage. Two objects are *totally associated* if they have the same storage sequence; two objects are *partially associated* if they share some storage but are not totally associated.

`EQUIVALENCE` statement
5.10.3

To indicate that two or more variables will share storage, they can be placed in an equivalence group in an `EQUIVALENCE` statement. If the objects in an equivalence group have different types or type parameters, no conversion or mathematical relationship is implied. If a scalar and an array are equivalenced, the scalar does not have array properties and the array does not have the properties of a scalar. The format of the `EQUIVALENCE` statement is defined as follows:

R545	*equivalence_stmt*	**is**	`EQUIVALENCE` *equivalence_set_list*
R546	*equivalence_set*	**is**	(*equivalence_object* **,** *equivalence_object_list*)
R547	*equivalence_object*	**is**	*variable_name*
		or	*array_element*
		or	*substring*

An equivalence object must not be one of the following items:

- A dummy argument
- A Fortran 90 pointer
- An allocatable array
- A structure containing a pointer at any level
- An automatic data object
- A function name, result name, or entry name
- A named constant
- A structure component
- A Cray pointee
- A subobject of any of the preceding

An equivalence group list must contain at least two items.

Any subscripts and subscript ranges must be integer initialization expressions (R504).

If an equivalence object is of type default integer, default real, double-precision real, default complex, default logical, or numeric sequence type, all objects in the set must be of these types.

If an equivalence object is of type character or character sequence type, all objects in the set must be type character. The lengths do not need to be the same.

If an equivalence object is of sequence type other than numeric or character sequence type, all objects in the set must be of the same type.

ANSI/ISO: The CF90 compiler allows equivalencing of character data with noncharacter data. The Fortran 90 standard does not address this. It is recommended that you do not perform equivalencing in this manner, however , because alignment and padding differs across platforms, thus rendering your code less portable.

If an equivalence object is of intrinsic type other than default integer, default real, double-precision real, default complex, or default logical, all objects in the set must be of the same type with the same kind type parameter value.

The use of an array name unqualified by a subscript list in an equivalence set specifies the first element of the array; that is, `A` means the first element of `A`.

An `EQUIVALENCE` statement must not specify that the same storage unit is to occur more than once in a storage sequence. For example, the following is illegal because it would indicate that storage for `X(2)` and `X(3)` is shared:

```
EQUIVALENCE (A, X(2)), (A, X(3))
```

An `EQUIVALENCE` statement must not specify the sharing of storage units between objects declared in different scoping units.

An `EQUIVALENCE` statement specifies that the storage sequences of the data objects in an equivalence set are storage associated. Any nonzero-sized sequences in the set have the same first storage unit, and any zero-sized sequences are storage associated with one another and with the first storage unit of any nonzero-sized sequences. This causes storage association of the objects in the group and may cause storage association of other data objects.

Example 1: The following code causes the alignment illustrated in Figure 7:

```
CHARACTER(LEN = 4) :: A, B
CHARACTER(LEN = 3) :: C(2)
EQUIVALENCE (A, C(1)), (B, C(2))
```

A(1:1)	A(2:2)	A(3:3)	A(4:4)			
			B(1:1)	B(2:2)	B(3:3)	B(4:4)
C(1)(1:1)	C(1)(2:2)	C(1)(3:3)	C(2)(1:1)	C(2)(2:2)	C(2)3:3)	

Figure 7. Character alignment example

As a result, the fourth character of A, the first character of B, and the first character of `C(2)` all share the same character storage unit.

Example 2: Figure 8 illustrates alignment of the following two numeric arrays:

```
REAL, DIMENSION(6) :: X, Y
EQUIVALENCE (X(5), Y(3))
```

X(1)	X(2)	X(3)	X(4)	X(5)	X(6)		
		Y(1)	Y(2)	Y(3)	Y(4)	5(5)	Y(6)

Figure 8. Numeric array alignment example

COMMON statement
5.10.4

The `COMMON` statement establishes blocks of storage called *common blocks* and specifies objects that are contained in the blocks. Two or more program units can share this space and thus share the values of variables stored in the space. Thus, the `COMMON` statement provides a global data facility based on storage association. Common blocks can be named, in which case they are called *named common blocks,* or they can be unnamed, in which case they are called *blank common.*

FORTRAN 77 restricted a common block to contain only numeric storage units or to contain only character storage units. Fortran 90 relaxes this restriction. Common blocks can contain mixtures of storage units and can contain unspecified storage units; however, if a common block contains a mixture of storage units, every declaration of the common block in the program must contain the same sequence of storage units, thereby matching types, kind type parameters, and attributes (`DIMENSION` and `POINTER`). The format of the `COMMON` statement is defined as follows:

R548	*common_stmt*	**is**	`COMMON` [/ [*common_block_name*] /] ≡ ≡ *common_block_object_list* ≡ ≡ [[,] / [*common_block_name*] / ≡ ≡ *common_block_object_list*] ...
R549	*common_block_object*	**is**	*variable_name* [(*explicit_shape_spec_list*)]

A *common_block_object* must not be one of the following items:

- A dummy argument
- An allocatable array
- An automatic object
- A function name, result name, or entry name
- A nonsequence structure

The appearance of two slashes with no common block name between them declares that the objects that follow are in blank common.

A common block name or an indication of blank common can appear more than once in one or more `COMMON` statements in the same scoping unit. The object list following each successive block name or blank common indication is treated as a continuation of the previous object list.

A variable can appear in only one common block within a scoping unit.

If a variable appears with an explicit-shape specification list, it is an array, and each bound must be a constant specification expression.

A nonpointer object of type default integer, default real, double-precision real, default complex, default logical, or numeric sequence type must become associated with only nonpointer objects of these types.

A nonpointer object of type character or character sequence must become associated with only nonpointer objects of these types.

If an object of numeric sequence or character sequence type appears in a common block, it is as if the individual components were enumerated in order directly in the common block object list.

A nonpointer object of sequence type other than numeric or character sequence type must become associated only with nonpointer objects of the same type.

A nonpointer object of intrinsic type other than default integer, default real, double precision real, default complex, default logical, or character must become associated only with nonpointer objects of the same type with the same kind type parameter value.

A pointer must become associated only with pointers of the same type, type parameters, and rank.

Only a named common block can be saved. Individual variables in the common block cannot be saved.

For each common block, a common block storage sequence is formed. It consists of the sequence of storage units of all the variables listed for the common block in the order of their appearance in the common block list. The storage sequence may be extended (on the end) to include the storage units of any variable equivalenced to a variable in the common block. Data objects storage associated with a variable in a common block are considered to be in that common block. The size of a common block is the size of its storage sequence including any extensions of the sequence resulting from equivalence association.

Within an executable program, the common block storage sequences of all nonzero-sized common blocks with the same name have the same first storage unit. Zero-sized common blocks are permitted. All zero-sized common blocks with the same name are storage associated with one another . The same is true for all blank common blocks except that because they can be of different sizes, it is possible for a zero-sized blank common block in one scoping unit to be associated with the first storage unit of a nonzero-sized blank common block in another scoping unit. In this way, many subprograms can use the same storage. They can specify common blocks to communicate global values or to reuse and conserve storage.

The CF90 compiler treats named common blocks and blank common blocks identically , as follows:

- Variables in blank common and variables in named common blocks can be initialized.
- Named common blocks and blank common are always saved.
- Named common blocks of the same name and blank common can be of different sizes in different scoping units.

ANSI/ISO: The Fortran 90 standard lists the following differences between blank common and named common blocks:

– Variables in blank common must not be initially defined in type declaration statements or `DATA` statements.

– A named common block is not saved unless it is named in a `SAVE` statement.

– Named common blocks of the same name must be of the same size in all scoping units.

Consider the code in the following example:

```
SUBROUTINE FIRST
   REAL B(2)
   COMPLEX C
   LOGICAL FLAG
   TYPE COORDINATES
      SEQUENCE
      REAL X, Y
      LOGICAL Z_O      ! ZERO ORIGIN?
   END TYPE COORDINATES
   TYPE(COORDINATES) P
   COMMON /REUSE/ B, C, FLAG, P

   REAL MY_VALUES(100)
   CHARACTER(LEN = 20) EXPLANATION
   COMMON /SHARE/ MY_VALUES, EXPLANATION
   SAVE /SHARE/

   REAL, POINTER :: W(:, :)
   REAL, TARGET, DIMENSION(100, 100) :: EITHER, OR
   INTEGER(SHORT) :: M(2000)
   COMMON /MIXED/ W, EITHER, OR, M
      . . .
END SUBROUTINE
SUBROUTINE SECOND
   INTEGER, PARAMETER :: SHORT = 2
   INTEGER I(8)
   COMMON /REUSE/ I

   REAL MY_VALUES(100)
   CHARACTER(LEN = 20) EXPLANATION
   COMMON /SHARE/ MY_VALUES, EXPLANATION
   SAVE /SHARE/

   REAL, POINTER :: V(:)
   REAL, TARGET, DIMENSION(100000) :: ONE, ANOTHER
   INTEGER(SHORT) :: M(2000)
   COMMON /MIXED/ V, ONE, ANOTHER, M  ! ILLEGAL
   . . .
END SUBROUTINE
```

Common block `REUSE` has a storage sequence of 8 numeric storage units. It is used to conserve storage. The storage referenced in subroutine `FIRST` is associated with the storage referenced in subroutine `SECOND`, as follows:

B(1)	B(2)	C		FLAG	X	Y	Z_O
I(1)	I(2)	I(3)	I(4)	I(5)	I(6)	I(7)	I(8)

Figure 9. Storage of `REUSE` in `FIRST` and `SECOND`

The Fortran 90 standard does not guarantee that the storage is actually retained and reused because, in the absence of a `SAVE` attribute for `REUSE`, some compilers can release the storage when either of the subroutines completes execution. The CF90 compiler treats named common blocks as entities that are contained in a `SAVE` statement.

Common block `SHARE` contains both numeric and character storage units and is used to share data between subroutines `FIRST` and `SECOND`.

The declaration of common block `MIXED` in subroutine `SECOND` is illegal because it does not have the same sequence of storage units as the declaration of `MIXED` in subroutine `FIRST`. The array pointer `W` in `FIRST` has two dimensions; the array pointer `V` in `SECOND` has only one. With common blocks, it is the sequence of storage units that must match, not the names of variables.

Restrictions on common and equivalence

5.10.5

An `EQUIVALENCE` statement must not cause two different common blocks to become associated and must not cause a common block to be extended by adding storage units preceding the first storage unit of the common block. For example, the following code is legal and results in the alignment illustrated in Figure 10:

```
COMMON A(5)
REAL B(5)
EQUIVALENCE (A(2), B(1))
```

A(1)	A(2)	A(3)	A(4)	A(5)	
	B(1)	B(2)	B(3)	B(4)	B(5)

Figure 10. Alignment resulting from correct code

On the other hand, the following code is not legal because it would place `B(1)` ahead of `A(1)`, as is illustrated in Figure 11:

```
EQUIVALENCE (A(1), B(2))
```

	A(1)	A(2)	A(3)	A(4)	A(5)
B(1)	B(2)	B(3)	B(4)	B(5)	

Figure 11. Alignment resulting from incorrect code

`COMMON` and `EQUIVALENCE` statements can appear in a module. If a common block is declared in a module, it must not also be declared in a scoping unit that accesses the module. The name of a `PUBLIC` data object from a module must not appear in a `COMMON` or `EQUIVALENCE` statement in any scoping unit that has access to the data object.

> **ANSI/ISO:** The Fortran 90 standard does not allow specification of a common block that is accessible through host or use association.

Using Data [6]

Section 5, which begins on page 131, explains how data objects are created and how their attributes are specified. This section explains how these objects can be used. The appearance of the name or designator where its value is required is a *reference* to the object. When an object is referenced, it must be defined; that is, it must have a value. The reference makes use of the value.

```
A = 1.0
B = A + 4.0
```

In the first statement, the constant value `1.0` is assigned to the variable `A`. It does not matter whether `A` was previously defined with a value or not; it now has a value and can be referenced in an executable statement. In the second statement, `A` is referenced; its value is obtained and added to the constant `4.0` to obtain a value that is then assigned to the variable `B`. The appearances of `A` in the first statement and `B` in the second statement are not considered to be references because their values are not required. The appearance of `A` in the second statement is a reference.

A data object can be a constant or a variable. Variables and constants can be scalar objects (with a single value) or arrays (with any number of values, all of the same type). Strictly speaking, there is no such thing as an array constant. An array constructor made up of all constant values is a *constant expression* , not an array constant. Also note that a derived type is a scalar value. Similar to an array constructor, a derived type constructor composed of all constant values is a constant expression, not a structure constant.

Arrays are said to be dynamic if their size can change. Automatic arrays are discussed in subsection 5.8, page 187; they are created on entry to a procedure, and their sizes are determined at that time. Allocatable arrays or pointer arrays can change size as well. The declared rank cannot change, but the extents of the dimensions may change with each reallocation or pointer assignment.

If a variable or constant is a portion of another object, it is called a *subobject* . A subobject can be one of the following items:

- An array element
- An array section
- A structure component
- A substring

A variable is referenced by its name, whereas a subobject is referenced by a *designator* . A designator indicates the portion of an object that is being referenced. Each subobject is considered to have a parent and is a portion of the parent. Each of the subobjects is described in this section.

This section also explains how to create and release pointers and allocatable arrays by using the `ALLOCATE` and `DEALLOCATE` statements. In addition, you can disassociate pointers from any target object by using the `NULLIFY` statement.

A reference to a variable or subobject is called a *data reference*. Guidelines exist for determining whether a particular data reference is classified as a character string, character substring, structure component, array, array element, or array section. These classifications are perhaps of more interest to compiler writers than to users of the language, but knowing how a data reference is classified makes it clearer which rules and restrictions apply to the reference, and easier to understand some of the explanations for the formation of expressions. Briefly, character strings and substrings must be of type character. Arrays have the `DIMENSION` attribute. Some data references can be classified as both structure components and array sections. In general, if a data reference contains a percent sign (`%`), it is a structure component, but its actual classification can be determined by other factors such as a section subscript or the rightmost element of the reference. If a substring range appears in a data reference, it must appear at the right end of the reference; the reference is considered to be a substring unless some component of the reference is an array section, in which case the data reference is considered to be an array section that just happens to have elements that are substrings. For a component reference to be classified as an array element, every component must have rank zero and a subscript list must appear at the right end of the reference. Subsections 6.1 through 6.4.5, page 205 through 215, contain many examples that demonstrate how these guidelines for classification apply.

Constants and variables

6.1

A constant has a value that cannot change; it can be a literal constant or a named constant (parameter). As explained in section 4, page 73, each of the intrinsic types has a form that specifies the type, type parameters, and value of a literal constant of the type. For user-defined types, there is a structure constructor to specify values of the type. If all of the components of a structure constructor are constants, the resulting derived-type value is a constant expression. Array constructors are used to form array values of any intrinsic or user-defined type. If all array elements are constant values, the resulting array is a constant array expression. A reference to a constant is always permitted.

A variable has a name such as `A` or a designator such as `B(I)`, and may or may not have a value. If it does not have a value, it must not be referenced. Variables are defined as follows:

R601	*variable*	**is**	*scalar_variable_name*
		or	*array_variable_name*
		or	*subobject*
R602	*subobject*	**is**	*array_element*
		or	*array_section*
		or	*structure_component*
		or	*substring*
R603	*logical_variable*	**is**	*variable*
R604	*default_logical_variable*	**is**	*variable*
R605	*char_variable*	**is**	*variable*
R606	*default_char_variable*	**is**	*variable*
R607	*int_variable*	**is**	*variable*
R608	*default_int_variable*	**is**	*variable*

A *logical_variable* must be of type logical, and a *default_logical_variable* must be of type default logical. A *char_variable* must be of type character and a *default_char_variable* must be of type default character. The CF90 compiler does not support any nondefault character types. An *int_variable* must be of type integer and a *default_int_variable* must be of type default integer .

Variables can be of any type, except for Boolean (typeless). There are contexts in which a variable must be of a certain type. In some of these cases, terms, such as *logical_variable* , *character_variable* , or Cray pointer, provide precise limitations.

ANSI/ISO: The Fortran 90 standard does not specify Boolean (typeless) constants or Cray pointers.

A subobject with a constant parent is not a variable.

A single object of any of the intrinsic or user-defined types is a *scalar* . A set of scalar objects, all of the same type and type parameters, can be arranged in a pattern involving columns, rows, planes, and higher-dimensioned configurations to form an array. An array has a rank between one and seven. A scalar has rank zero. In simple terms, an array is an object with the DIMENSION attribute; a scalar is not an array. For example:

```
TYPE PERSON
   INTEGER AGE
   CHARACTER(LEN = 40) NAME
END TYPE PERSON

TYPE(PERSON) FIRECHIEF, FIREMEN(50)
CHARACTER(20) DISTRICT, STATIONS(10)
```

The following data references are classified as indicated by the comments on each line:

```
DISTRICT      ! character string
DISTRICT(1:6) ! substring
FIRECHIEF%AGE ! structure component
FIREMEN%AGE   ! array of integers
STATIONS      ! array of character strings
STATIONS(1)   ! array element (character string)
STATIONS(1:4) ! array section of character strings
```

The following code segment shows that a subobject can have a constant parent:

```
CHARACTER(*), PARAMETER :: MY_DISTRICT = "DISTRICT 13"
CHARACTER(2) DISTRICT_NUMBER
DISTRICT_NUMBER = MY_DISTRICT(10:11)
```

`DISTRICT_NUMBER` has the value `13`.

Substrings

6.2

A *character string* consists of zero or more characters. Even though it is made up of individual characters, a character string is considered to be scalar . As with any data type, it is possible to declare an array of character strings, all of the same length.

A *substring* is a contiguous portion of a character string that has a starting point and an ending point within the character string. It is possible to reference a substring of a character variable or constant.

Substrings are defined as follows:

R609	*substring*	**is**	*parent_string* (*substring_range*)
R610	*parent_string*	**is**	*scalar_variable_name*
		or	*array_element*
		or	*scalar_structure_component*
		or	*scalar_constant*
R611	*substring_range*	**is**	[*scalar_int_expr*] **:** [*scalar_int_expr*]

The *parent_string* of a substring must be of type character. The *substring* is of type character.

The *scalar_int_expr* at the left in the *substring_range* is the starting position. The *scalar_int_expr* at the right in the *substring_range* is the ending position.

A *substring* is the contiguous sequence of characters within the string, beginning with the character at the starting position and ending at the ending position. If the starting position is omitted, the default is 1; if the ending position is omitted, the default is the length of the character string.

The length of a character string or substring can be 0, but not a negative number . Zero-length strings result when the starting position is greater than the ending position. The formula for calculating the length of a string is as follows:

MAX (*ending_position* − *starting_position* + 1, 0)

The first character of a parent string is at position 1 and the last character is at position *n* where *n* is the length of the string. The starting position of a substring must be greater than or equal to 1 and the ending position must be less than or equal to the length *n*, unless the length of the substring is 0. If the parent string is of length 0, the substring must be of length 0.

Example 1:

```
CHARACTER(14) NAME
NAME = "John Q. Public"
NAME(1:4) = "Jane"
PRINT *, NAME(9:14)
```

In example 1, NAME is a scalar character variable, a string of 14 characters, that is assigned the value John Q. Public by the first assignment statement. NAME(1:4) is a substring of four characters that is reassigned the value Jane by the second assignment statement, leaving the remainder of the string NAME unchanged; the string NAME then becomes Jane Q. Public. The PRINT statement prints the characters in positions 9 through 14, in this case, the surname, Public.

Example 2:

Assume the following definition and declarations:

```
TYPE PERSON
   INTEGER AGE
   CHARACTER(LEN = 40) NAME
END TYPE PERSON

TYPE(PERSON) FIRECHIEF, FIREMEN(50)
CHARACTER(20) DISTRICT, STATIONS(10)
```

The following are all substrings:

```
STATIONS(1)(1:5)     ! array element as parent string
FIRECHIEF%NAME(4:9)  ! structure component as parent string
DISTRICT(7:14)       ! scalar variable as parent string
'0123456789'(N:N+1)  ! character constant as parent string
```

In example 2, the reference `STATIONS(:)(1:5)` is permitted. It is an array whose elements are substrings, but it is not considered to be a substring reference. Even though the entire array is indicated, this reference is considered to be an array section reference. The description is in subsection 6.4.5, page 215. `STATIONS(1:5)(1:5)` is also permitted. It is an array section whose elements are substrings. Whenever an array is constructed of character strings and any part of it (other than the whole object) is referenced, an array section subscript must appear before the substring range specification, if any. Otherwise, the substring range specification will be treated as an array section specification because the two have the same form. `STATIONS(1:5)` is an array section reference that references the entire character strings of the first five elements of `STATIONS`. The last line of the example is a substring where the parent is a constant and the starting and ending positions are variable.

Structure components

6.3

A *structure* is an aggregate of components of intrinsic or derived types. It is itself an object of derived type. The types and attributes of the components are specified in the type definition; they can be scalars or arrays. Each structure has at least one component. There can be arrays of structures. In example 2 (see subsection 6.2, page 207), `FIRECHIEF` is a structure; `FIREMEN` is an array of structures of type `PERSON`.

A component of a structure is referenced by placing the name of the component after the name of the parent structure, separated by a percent sign (`%`). For example, `FIRECHIEF % NAME` references the character string component of the variable `FIRECHIEF` of type `PERSON`.

A structure component is a data reference and is defined as follows:

R612	*data_ref*	**is**	*part_ref* [`%` *part_ref*] ...
R613	*part_ref*	**is**	*part_name* [(*section_subscript_list*)]
R614	*structure_component*	**is**	*data_ref*

For a data reference to be considered a structure component reference, there must be more than one part reference and the rightmost part reference must be a part name. If the rightmost component is of the following form, the reference is considered to be an array section or array element (the simplest form of a section subscript list is a subscript list):

part_name (*section_subscript_list*)

The rules for forming a *section_subscript_list* and a *subscript* are provided in subsection 6.4, page 212.

In a data reference, each part name except the rightmost must be of derived type.

In a data reference, each part name except the leftmost must be the name of a component of the derived-type definition of the type of the preceding part name.

In a part reference containing a section subscript list, the number of section subscripts must equal the rank of the part name.

It is possible to create a structure with more than one array part, but in a data reference to the structure, there must not be more than one part reference with nonzero rank.

In a data reference, a part name to the right of a part reference with nonzero rank must not have the `POINTER` attribute. It is possible to declare an array of structures that have a pointer as a component, but it is not possible to reference such an object as an array.

The rank of a part reference consisting of just a part name is the rank of the part name. The rank of a part reference of the following form is the number of subscript triplets and vector subscripts in the list:

part_name (*section_subscript_list*)

The rank is less than the rank of the part name if any of the section subscripts are subscripts other than subscript triplets or vector subscripts. The shape of a data reference is the shape of the part reference with nonzero rank, if any; otherwise, the data reference is a scalar and has rank zero.

The *parent structure* in a data reference is the data object specified by the leftmost part name. If the parent object has the `INTENT`, `TARGET`, or `PARAMETER` attribute, the structure component has the attribute. The type and type parameters of a structure component are those of the rightmost part name. A structure component is a pointer only if the rightmost part name has the `POINTER` attribute. Typically, an object cannot have both the `TARGET` and `POINTER` attributes. However, a structure that has the `TARGET` attribute can contain components that have the `POINTER` attribute.

Example 1: Assume the following type definition and structure declarations:

```
TYPE PERSON
   INTEGER AGE
   CHARACTER(LEN = 40) NAME
END TYPE PERSON

TYPE(PERSON) FIRECHIEF, FIREMEN(50)
```

In this example, structure components are as follows:

```
FIRECHIEF%AGE    ! scalar component of scalar parent
FIREMEN(J)%NAME  ! component of array element parent
FIREMEN(1:N)%AGE ! component of array section parent
```

Example 2: If a derived-type definition contains a component that is of derived type, then a reference to an ultimate component can contain more than two part references as do the references in the first two PRINT statements in the following example.

```
TYPE REPAIR_BILL
   REAL PARTS
   REAL LABOR
END TYPE REPAIR_BILL

TYPE VEHICLE
   CHARACTER(LEN = 40) OWNER
   INTEGER MILEAGE
   TYPE(REPAIR_BILL) COST
END TYPE VEHICLE

TYPE(VEHICLE) BLACK_FORD, RED_FERRARI

PRINT *, BLACK_FORD%COST%PARTS
PRINT *, RED_FERRARI%COST%LABOR
PRINT *, RED_FERRARI%OWNER
```

Arrays

6.4

An *array* is a collection of scalar elements of any intrinsic or derived type. An object of any type that is specified to have the DIMENSION attribute is an array . All of the elements of an array must have the same type and kind parameter . There can be arrays of structures. The value returned by a function can be an array. The appearance of an array name or designator has no implications for the order in which the individual elements are referenced unless array element ordering is specifically required.

Array terminology

6.4.1

An array consists of elements that extend in one or more dimensions to represent columns, rows, planes, and so on. There can be up to seven dimensions in an array declaration. The number of dimensions in an array is called the *rank* of the array. The number of elements in a dimension is called the *extent* of the array in that dimension. The *shape* of an array is determined from the rank and the extents; to be precise, the shape is a vector where each element of the vector is the extent in the corresponding dimension. The *size* of an array is the product of the extents; that is, it is the total number of elements in the array.

Example 1: In the following example, the rank of `X` is `2` because `X` has two dimensions. The extent of the first dimension is `10`; the extent of the second dimension is `2`. The shape of `X` is `10` by `2`, that is, a vector of two values, `(10, 2)`. The size is `20`, the product of the extents.

```
REAL X(0:9, 2)
```

An object can be given the `DIMENSION` attribute in a type declaration or in one of several other specification statements.

Example 2: The following example statements show some ways of declaring that `A` has rank `3` and shape `(10, 15, 3)`:

```
DIMENSION A(10, 15, 3)
REAL, DIMENSION(10, 15, 3) :: A
REAL A(10, 15, 3)
COMMON A(10, 15, 3)
TARGET A(10, 15, 3)
```

Arrays of nonzero size have a lower and upper bound for each dimension. The *lower bound* is the smallest subscript value for a dimension; the *upper bound* is the largest subscript value for that dimension. The default lower bound is 1 if the lower bound is omitted in the declaration. Array bounds can be positive, zero, or negative.

Example 3: In the following example, the first dimension of `Z` ranges from `−3` to `10`, that is, `−3`, `−2`, `−1`, `0`, `1`, `2`, . . ., `9`, `10`. The lower bound is `−3`; the upper bound is `10`. In the second dimension, the lower bound is `1`; the upper bound is `12`.

```
REAL Z(−3:10, 12)
```

The bounds for array expressions are described in subsection 7.2.8.4, page 276.

Whole arrays
6.4.2

Some arrays are named. The name is either an array variable name or the name of a constant. If the array name appears without a subscript list or section subscript list, all of the elements of the array are referenced and the reference is considered to be a *whole array* reference.

Array elements
6.4.3

An *array element* is one of the scalar elements that make up an array. A subscript list is used to indicate which element is referenced. Assume that A is declared to be a one-dimensional array, as follows:

```
REAL, DIMENSION(10) :: A
```

A(1) refers to the first element, A(2) to the second, and so on. The number in the parentheses is the subscript that indicates which scalar element is referenced. Assume that B is declared to be a seven-dimensional array, as follows:

```
REAL B(5, 5, 5, 5, 4, 7, 5)
```

B(2,3,5,1,3,7,2) refers to one scalar element of B, indexed by a subscript in each dimension. The set of numbers that indicate the position along each dimension in turn (in this case, 2,3,5,1,3,7,2) is called a *subscript list*.

Array sections
6.4.4

Sometimes only a portion of an array is needed for a calculation. It is possible to refer to a selected portion of an array as an array; this portion is called an *array section*. A *parent array* is the whole array from which the portion that forms the array section is selected.

An array section is specified by an array variable name and a section subscript list that consists of subscripts, triplet subscripts, or vector subscripts. At least one subscript must be a triplet or vector subscript; otherwise, the reference indicates an array element, not an array section. The following example uses a section subscript to create an array section:

```
REAL A(10)
    . . .
A(2:5) = 1.0
```

The parent array `A` has 10 elements. The array section consists of the elements `A(2)`, `A(3)`, `A(4)`, and `A(5)` of the parent array. The section `A(2:5)` is an array itself and the value `1.0` is assigned to all four of its elements.

Format of array elements and array sections

6.4.5

The format of an array element is a data reference (R612), which is defined as follows:

R615	*array_element*	**is**	*data_ref*

For a data reference to be classified as an array section, exactly one part reference must have nonzero rank, and either the final part reference must have a section subscript list with nonzero rank or another part reference must have nonzero rank.

The format of an array section (R616) is a data reference followed by an optional substring range enclosed in parentheses. This is defined as follows:

R616	*array_section*	**is**	*data_ref* [(*substring_range*)]

The format of a substring range is found in subsection 6.2, page 207.

A part name in a data reference can be followed by an optional section subscript list, as follows:

R617	*subscript*	**is**	*scalar_int_exp*
R618	*section_subscript*	**is**	*subscript*
		or	*subscript_triplet*
		or	*vector_subscript*
R619	*subscript_triplet*	**is**	[*subscript*] **:** [*subscript*] [**:** *stride*]
R620	*stride*	**is**	*scalar_int_expr*
R621	*vector_subscript*	**is**	*int_expr*

Each *subscript* and *stride* must be a *scalar_int_expr* . A *vector_subscript* must be an integer array expression of rank one.

For a data reference to be classified as an array element, every part reference must have rank zero and the last part reference must contain a subscript list.

In an array section that is a data reference followed by a substring range, the rightmost part name must be of type character.

In an array section of an assumed-size array, the second subscript must not be omitted from a subscript triplet in the last dimension.

A section subscript must be present for each dimension of an array. If any section subscript is simply a subscript, the section will have a lesser rank than its parent.

If any part of a reference is an array section, the reference is considered to be an array section reference. In a data reference, there may be at most one part with rank greater than zero.

Examples of array elements and array sections are as follows:

```
ARRAY_A(1,2)          ! array element
ARRAY_A(1:N:2,M)      ! rank-one array section
ARRAY_B(:,:,:)(2:3)   ! array whose elements are substrings
                      ! of length 2
SCALAR_A%ARRAY_C(L)                      ! array element
SCALAR_A%ARRAY_C(1:L)                    ! array section
SCALAR_B%ARRAY_D(1:N)%SCALAR_C           ! array section
ARRAY_E(1:N:2)%ARRAY_F(I,J)%STRING(K)(:) ! array section
```

In the last example above, each component of the type definition is an array and the object `ARRAY_E` is an array. The reference is valid because each component in the reference is scalar. The substring range is not needed because it specifies the entire string; however, it serves as a reminder that the last component is of type character.

The following examples demonstrate the allowable combinations of scalar and array parents with scalar and array components.

```
TYPE REPAIR_BILL
   REAL PARTS(20)
   REAL LABOR
END TYPE REPAIR_BILL

TYPE(REPAIR_BILL) FIRST
TYPE(REPAIR_BILL) FOR_1990(6)
```

Scalar parent:

```
1.FIRST % LABOR           ! structure component
2.FIRST % PARTS(I)        ! array element
3.FIRST % PARTS           ! component (array-valued)
4.FIRST % PARTS(I:J)      ! array section

5.FOR_1990(K) % LABOR     ! structure component
6.FOR_1990(K) % PARTS(I) ! array element
7.FOR_1990(K) % PARTS     ! component (array-valued)
8.FOR_1990(K) % PARTS(I:J) ! array section
```

Array parent:

```
9.FOR_1990 % LABOR        ! component and array section
10.FOR_1990 % PARTS(I)    ! array section
11.FOR_1990 % PARTS       ! ILLEGAL
12.FOR_1990 % PARTS(I:J) ! ILLEGAL

13.FOR_1990(K:L) % LABOR ! component and array section
14.FOR_1990(K:L) % PARTS(I) ! array section
15.FOR_1990(K:L) % PARTS ! ILLEGAL
16.FOR_1990(K:L) % PARTS(I:J) ! ILLEGALText goes here
```

References `11`, `12`, `15`, and `16` are illegal because only one component may be of rank greater than zero. References `3` and `7` are compact (contiguous) array objects and are classified as array-valued structure components. References `9`, `10`, `13`, and `14` are noncontiguous array objects and are classified as sections. These distinctions are important when such objects are actual arguments in procedure references.

Subscripts 6.4.5.1

In an array element reference, each subscript must be within the bounds for that dimension. A subscript can appear in an array section reference. Whenever this occurs, it decreases the rank of the section by one less than the rank of the parent array. A subscript used in this way must be within the bounds for the dimension. The CF90 compiler allows overindexing, which may produce incorrect results because of optimization. For information on overindexing and optimization, see the *CF90 Fortran Language Reference Manual, Volume 3*, publication SR–3905.

ANSI/ISO: The Fortran 90 standard does not address overindexing.

Subscript triplets 6.4.5.2

The first subscript in a subscript triplet is the lower bound; the second is the upper bound. If the lower bound is omitted, the declared lower bound is used. If the upper bound is omitted, the declared upper bound is used. The *stride* is the increment between successive subscripts in the sequence. If it is omitted, it is assumed to be 1. The stride must not be 0. If the subscripts and stride are omitted and only the colon (`:`) appears, the entire declared range for the dimension is used.

When the stride is positive, an increasing sequence of integer values is specified from the first subscript in increments of the stride, up to the last value that is not greater than the second subscript. The sequence is empty if the first subscript is greater than the second. If any subscript sequence is empty , the array section is a zero-sized array, because the size of the array is the product of its extents. For example, given the array declared `A(5,4,3)` and the section `A(3:5,2,1:2)`, the array section is of rank `2` with shape `(3,2)` and size `6`. The elements are as follows:

```
A(3,2,1)     A(3,2,2)
A(4,2,1)     A(4,2,2)
A(5,2,1)     A(5,2,2)
```

When the stride is negative, a decreasing sequence of integer values is specified from the first subscript, in increments of the stride, down to the last value that is not less than the second subscript. The sequence is empty if the second subscript is greater than the first, and the array section is a zero-sized array. For example, given the array declared `B(10)` and the section `B (9:4:-2)`, the array section is of rank `1` with shape `(3)` and size `3`. The elements are as follows:

```
B(9) B(7) B(5)
```

However , array section `B(9:4)` is a zero-sized array.

A subscript in a subscript triplet is not required to be within the declared bounds for the dimension as long as all subscript values selected by the triplet are within the declared bounds. For example, given an array declared `B(10)`, section `B(3:11:7)` is permitted. It has rank `1` with shape `(2)` and size `2`. The elements are as follows:

```
B(3) B(10)
```

Vector subscripts
6.4.5.3

While subscript triplets specify values in increasing or decreasing order with a specified stride to form a regular pattern, vector subscripts specify values in arbitrary order. The values must be within the declared bounds for the dimension. A vector subscript is a rank-one array of integer values used as a section subscript to select elements from a parent array.

```
INTEGER J(3)
REAL A(30)
   . . .
J = (/ 8, 4, 7 /)
A(J) = 1.0
```

The last assignment statement assigns the value `1.0` to `A(4)`, `A(7)`, and `A(8)`. The section `A(J)` is a rank-one array with shape `(3)` and size `3`.

If `J` were assigned `(/ 4, 7, 4 /)` instead, the element `A(4)` would be accessed in two ways: as `A(J(1))` and as `A(J(3))`. Such an array section is called a many-to-one array section. A many-to-one section must not appear on the left of the equal sign in an assignment statement or as an input item in a `READ` statement. The reason is that the result will depend on the order of evaluation of the subscripts, which is not specified by the language. The results would not be predictable and the program containing such a statement would not be portable.

Array sections with vector subscripts are array expressions, so there are places, such as the following, where array sections with vector subscripts must not appear:

- As internal files
- As pointer targets
- As actual arguments for `INTENT(OUT)` or `INTENT(INOUT)` dummy arguments

Using array elements and array sections
6.4.6

Subscripts, subscript triplets, and vector subscripts can be mixed in a single section subscript list used to specify an array section. A triplet section can specify an empty sequence (for example `1:0`), in which case the resulting section is a zero-sized array.

Example 1: Assume that `B` is declared as follows:

```
REAL B(10, 10, 5)
```

Section `B(1:4:3, 6:8:2, 3)` consists of the following four elements:

```
B(1,6,3)      B(1,8,3)
B(4,6,3)      B(4,8,3)
```

The stride along the first dimension is 3, resulting in a subscript-value list of 1 and 4. The stride along the second dimension is 2, resulting in a subscript-value list of 6 and 8. In the third position there is a subscript that reduces the rank of the section by 1. The section has shape (2, 2) and size 4.

Example 2: Assume IV is declared as follows:

```
INTEGER, DIMENSION(3) :: IV = (/ 4, 5, 4 /)
```

Then the section B(8:9, 5, IV) is a 2 x 3 array consisting of the following six elements:

```
B(8,5,4)     B(8,5,5)     B(8,5,4)
B(9,5,4)     B(9,5,5)     B(9,5,4)
```

B(8:9, 5:4, IV) is a zero-sized array of rank 3.

Array element order

6.4.7

When whole arrays are used as operands in an executable statement, the indicated operation is performed element-by-element, but no order is implied for these elemental operations. They can be executed in any order or simultaneously. Although there is no order of evaluation when whole array operations are performed, there is an ordering of the elements in an array itself. An ordering is required for the input and output of arrays and for certain intrinsic functions such as MAXLOC. The elements of an array form a sequence whose ordering is called *array element order*. This is the sequence that occurs when the subscripts along the first dimension vary most rapidly, and the subscripts along the last dimension vary most slowly. Thus, for an array declared as REAL A(3, 2), the elements in array element order are: A(1, 1), A(2, 1), A(3, 1), A(1, 2), A(2, 2), A(3, 2).

The position of an array element in this sequence is its *subscript order value*. Element A(1, 1) has a subscript order value of 1. Element A(1, 2) has a subscript order value of 4. Table 8 shows how to compute the subscript order value for any element in arrays of rank 1 through 7.

Table 8. Computation of subscript order value

Rank	Explicit shape specifier	Subscript list	Subscript order value
1	j_1:k_1	s_1	$1 + (s_1 - j_1)$
2	j_1:k_1, j_2:k_2	s_1, s_2	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$
3	j_1:k_1, j_2:k_2, j_3:k_3	s_1, s_2, s_3	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$
.	.	.	.
.	.	.	.
.	.	.	.
7	j_1:k_1, . . ., j_7:k_7	s_1, . . ., s_7	$1 + (s_1 - j_1)$ $+ (s_2 - j_2) \times d_1$ $+ (s_3 - j_3) \times d_2 \times d_1$ $+ \ldots$ $+ (s_7 - j_7) \times d_6$ $\times d_5 \times \ldots \times d_1$

Notes:
$d_i = \max (k_i - j_i + 1, 0)$ is the size of the ith dimension.
If the size of the array is nonzero, $j_i \leq s_i \leq k_i$ for all $i = 1, 2, \ldots, 7$.

The subscript order of the elements of an array section is that of the array object that the section represents. That is, given the array `A(10)` and the section `A(2:9:2)` consisting of the elements `A(2)`, `A(4)`, `A(6)`, and `A(8)`, the subscript order value of `A(2)` in the array section `A(2:9:2)` is `1`; the subscript order value of `A(4)` in the section is `2` and `A(8)` is `4`.

Pointers and allocatable arrays

6.5

Fortran 90 provides several dynamic data objects. Automatic objects (arrays and character strings) are discussed in subsection 5.8, page 187. In addition, there are two data attributes that can be used to specify dynamic data objects: `ALLOCATABLE` and `POINTER`. Arrays of any type can have the `ALLOCATABLE` attribute; scalars or arrays of any type can have the `POINTER` attribute. Section 5, page 131, described how such objects are declared. This section describes how space is created for these objects with the `ALLOCATE` statement, how it can be released with the `DEALLOCATE` statement, and how pointers can be disassociated from any target with the `NULLIFY` statement. The association status of a pointer can be defined or undefined; initially (when a pointer is declared), it is undefined. If it is defined, the pointer can be associated with a target or disassociated from any target. The target is referenced by the name of the pointer and is like any other variable in that it is defined when it acquires a value. Figure 12 shows the various states that a pointer may assume.

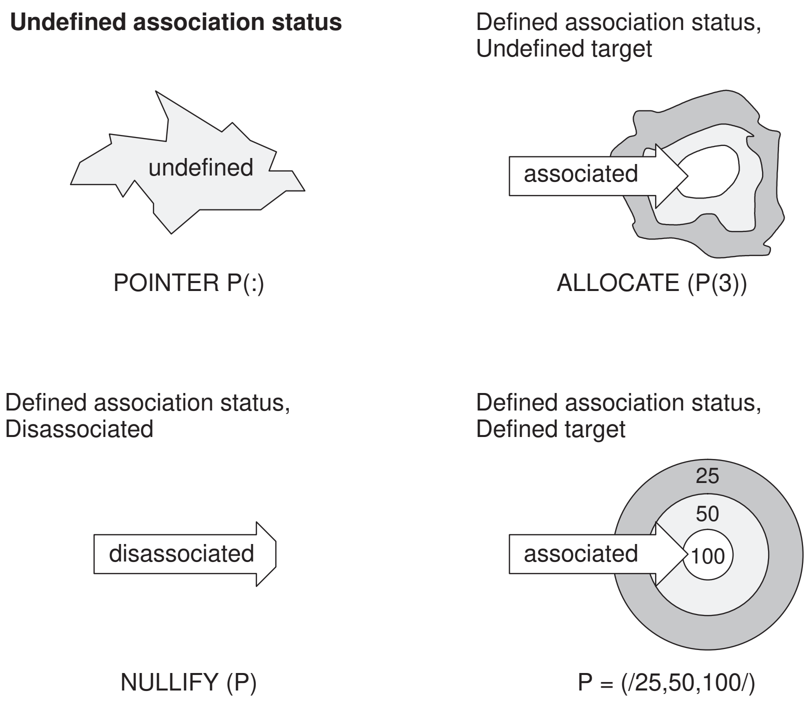

Figure 12. States in the lifetime of a pointer

Subsection 7.5.3, page 313, describes how pointers can be associated with existing space and how dynamic objects can acquire values.

6.5.1 `ALLOCATE` statement

The `ALLOCATE` statement creates space for the following:

- Arrays with the `ALLOCATABLE` attribute
- Variables with the `POINTER` attribute

The `ALLOCATE` statement is defined as follows:

R622	*allocate_stmt*	**is**	`ALLOCATE` (*allocation_list* ≡ ≡ [, `STAT` = *stat_variable*])
R623	*stat_variable*	**is**	*scalar_int_variable*
R624	*allocation*	**is**	*allocate_object* [(*allocate_shape_spec_list*)]
R625	*allocate_object*	**is**	*variable_name*
		or	*structure_component*
R626	*allocate_shape_spec*	**is**	[*allocate_lower_bound* **:**] *allocate_upper_bound*
R627	*allocate_lower_bound*	**is**	*scalar_int_expr*
R628	*allocate_upper_bound*	**is**	*scalar_int_expr*

The *stat_variable*, *allocate_lower_bound*, and *allocate_upper_bound* must each be scalar integer expressions.

Each *allocate_object* must be a pointer or an allocatable array.

An attempt to allocate space for an allocatable array that is currently allocated results in an error condition.

If a `STAT=` variable appears, it must not be allocated in the same `ALLOCATE` statement. It is set to zero if the allocation is successful and is set to a positive value if there is an error condition; when an error is detected, subsequent items in the *allocation_list* are not allocated. If there is no `STAT=` variable, the program terminates when an error condition occurs.

An argument to an inquiry function in an `ALLOCATE` statement must not appear as an allocate object in that statement. For example, the use of the intrinsic inquiry function `SIZE` in the following code is not permitted.

```
REAL, ALLOCATABLE :: A(:), B(:)
ALLOCATE (A(10), B(SIZE(A)))
```

The number of allocate shape specifications must agree with the rank of the array .

If the lower bound is omitted, the default is 1. If the upper bound is less than the lower bound, the extent in that dimension is 0 and the array has zero size.

An allocate object can be of type character and it can have a length of 0, in which case no memory is allocated.

The values of the bounds expressions at the time an array is allocated determine the shape of the array. If an entity in a bounds expression is subsequently redefined, the shape of the allocated array is not changed.

Allocation of allocatable arrays 6.5.1.1

You must declare the rank of an allocatable array, but the bounds, extents, shape, and size are determined when the array is allocated. After allocation the array can be defined and redefined. The array then is said to be *currently allocated*. It is an error to allocate an allocatable array that is already allocated. The intrinsic function ALLOCATED can be used to query the allocation status of an allocatable array if the allocation status is defined, for example:

```
REAL, ALLOCATABLE :: X(:, :, :)
    . . .
IF (.NOT. ALLOCATED(X)) ALLOCATE (X(-6:2, 10, 3))
```

X is not available for use in the program until it has been allocated space by an ALLOCATE statement. X must be declared with a deferred-shape array specification and the ALLOCATABLE attribute.

Allocation of pointers 6.5.1.2

When an object with the POINTER attribute is allocated, space is created, and the pointer is associated with that space, which becomes the pointer target. A reference to the pointer name can be used to define or access its target. The target can be an array or a scalar. Additional pointers can become associated with the same target by pointer assignment (described in subsection 7.5.3, page 313). A pointer target can be an array with the ALLOCATABLE attribute if the array also has the TARGET attribute. Allocation of a pointer creates an object that implicitly has the TARGET attribute.

It is not an error to allocate a pointer that is currently associated with a target. In this case, a new pointer target is created and the previous association of the pointer is lost. If there was no other way to access the previous target, it becomes inaccessible.

The ASSOCIATED intrinsic function can be used to query the association status of a pointer if the association status of the pointer is defined. The ASSOCIATED intrinsic function also can be used to inquire whether a pointer is associated with a target or whether two pointers are associated with the same target.

Pointers can be used in many ways; an important usage is creating linked lists. For example:

```
TYPE NODE
   INTEGER :: VALUE
   TYPE(NODE), POINTER :: NEXT
END TYPE NODE

TYPE(NODE), POINTER :: LIST
    . . .
ALLOCATE(LIST)
LIST%VALUE = 17
ALLOCATE(LIST%NEXT)
```

The first two executable statements create a node pointed to by LIST and put the value 17 in the VALUE component of the node. The third statement creates a second node pointed to by the NEXT component of the first node.

NULLIFY statement
6.5.2

The NULLIFY statement causes a pointer to be disassociated from any target. Pointers have an initial association status that is undefined. To initialize a pointer to point to no target, it is necessary to execute a NULLIFY statement for the pointer.

The NULLIFY statement is defined as follows:

R629	*nullify_stmt*	**is**	NULLIFY (*pointer_object_list*)
R630	*pointer_object*	**is**	*variable_name*
		or	*structure_component*

Each *pointer_object* must have the POINTER attribute.

6.5.3 DEALLOCATE statement

The DEALLOCATE statement releases the space allocated for an allocatable array or a pointer target and nullifies the pointer. After an allocatable array or pointer has been deallocated, it cannot be accessed or defined until it is allocated again or, in the case of a pointer, assigned to an existing target.

In some cases, the execution of a RETURN statement in a subprogram may cause the association status of a pointer to become undefined. This can be avoided if the pointer is given the SAVE attribute or if it is declared in a subprogram that remains active. (The main program is always active. Variables declared in modules accessed by the main program and named common blocks specified in the main program do not need to be given the SAVE attribute; these entities have the attribute automatically. If the main program calls subroutine A and subroutine A calls function B, then the main program, subroutine A, and function B are active until a return from function B is executed, at which time only the main program and subroutine A are active. If a recursive subprogram becomes active, it remains active until the return from its first invocation is executed.)

The DEALLOCATE statement is defined as follows:

R631	*deallocate_stmt*	is	DEALLOCATE (*allocate_object_list* ≡
			≡ [, STAT = *stat_variable*])

The *stat_variable* must be a scalar integer variable.

Each *allocate_object* must be a pointer or an allocatable array.

If there is a `STAT=` variable and it is a pointer, it must not be deallocated in the same `DEALLOCATE` statement. The *stat_variable* is set to zero if the deallocation is successful and is set to a positive value if there is an error condition. If `STAT=` *stat_variable* is not specified, the program terminates when an error condition occurs.

6.5.3.1 Deallocation of allocatable arrays

To be deallocated, an allocatable array must be currently allocated; otherwise, an error condition will occur. You can use the inquiry intrinsic function `ALLOCATED` to determine if an array is currently allocated.

An allocatable array can have the `TARGET` attribute. If such an array is deallocated, the association status of any pointer associated with the array will become undefined. Such an array must be deallocated by the appearance of its name in a `DEALLOCATE` statement. It must not be deallocated by the appearance of the pointer name in a `DEALLOCATE` statement.

According to the Fortran 90 standard, when a `RETURN` or `END` statement is executed in a subprogram, allocatable arrays become undefined and their allocation status becomes undefined unless any of the following conditions exist:

- The array has the `SAVE` attribute
- The array is specified in a module that is accessed by an active subprogram
- The array is accessed by host association

Any other allocatable arrays should be deallocated before leaving the subprogram because if an allocatable array acquires an undefined allocation status, it can no longer be referenced, defined, allocated, or deallocated.

Upon executing a `RETURN` or `END` statement, the CF90 compiler deallocates allocatable arrays that are not included in a `SAVE` statement. The CF90 compiler method of deallocating arrays may not be portable across implementations of Fortran 90.

The following is an example of the allocation and deallocation of an allocatable array:

```
REAL, ALLOCATABLE :: X(:, :)
   . . .
ALLOCATE (X(10, 2), STAT = IERR)
IF (IERR .GT. 0) CALL HANDLER
X = 0.0
   . . .
DEALLOCATE (X)
   . . .
ALLOCATE (X(-10:10, 5), STAT = JERR)
```

`X` is declared to be a deferred-shape, two-dimensional, real array with the `ALLOCATABLE` attribute. Space is allocated for it and it is given bounds, extents, shape, and size and then initialized to have zero values in all elements. Later `X` is deallocated, and still later, it is again allocated with different bounds, extents, shape, and size, but its rank remains as declared.

Deallocation of pointers
6.5.3.2

Only a pointer with defined association status can be deallocated. Deallocating a pointer with an undefined association status or a pointer associated with a target that was not created by allocation causes an error condition in the `DEALLOCATE` statement. A pointer associated with an allocatable array must not be deallocated. (Of course, the array itself can be deallocated.)

It is possible (by pointer assignment) to associate a pointer with a portion of an object such as an array section, an array element, or a substring. A pointer associated with only a portion of an object cannot be deallocated. If more than one pointer is associated with an object, deallocating one of the pointers causes the association status of the others to become undefined. Such pointers must not be arguments to the `ASSOCIATED` inquiry function.

When a `RETURN` or `END` statement is executed in a procedure, the association status of a pointer declared or accessed in the procedure becomes undefined unless one of the following conditions is true:

- The pointer has the `SAVE` attribute
- The pointer is specified in a module that is accessed by an active subprogram
- The pointer is accessed by host association
- The pointer is in blank common
- The pointer is in a named common block that is specified in an active subprogram or has the `SAVE` attribute
- The pointer is the return value of a function declared to have the `POINTER` attribute

If the association status of a pointer becomes undefined, the pointer can no longer be referenced, defined, or deallocated. It can be allocated, nullified, or pointer-assigned to a new target.

The following is an example of the allocation and deallocation of a pointer:

```
REAL, POINTER :: X(:, :)
    . . .
ALLOCATE (X (10, 2), STAT = IERR)
IF (IERR .GT. 0) CALL HANDLER
X = 0.0
    . . .
DEALLOCATE (X)
    . . .
ALLOCATE (X(-10:10, 5), STAT = JERR)
```

`X` is declared to be a deferred-shape, two-dimensional, real array with the `POINTER` attribute. Space is allocated for it and it is given bounds, extents, shape, and size and then initialized to have zero values in all elements. Later `X` is deallocated, and still later, it is again allocated with different bounds, extents, shape, and size. This example is quite similar to the previous example for allocatable arrays, except that, in the case of pointers, it is not necessary to deallocate `X` before allocating it again.

Expressions and Assignments [7]

In Fortran, calculations are specified by writing *expressions*. Expressions look much like algebraic formulas in mathematics, particularly when the expressions involve calculations on numerical values.

Expressions often involve nonnumeric values, such as character strings, logical values, or structures; these also can be considered to be formulas that involve nonnumeric quantities rather than numeric ones.

This section describes how valid expressions can be formed, how they are interpreted, and how they are evaluated. One of the major uses of expressions is in assignment statements where the value of an expression is assigned to a variable. The assignment statement appears in four forms: intrinsic assignment, defined assignment, masked array assignment, and pointer assignment. In the first three forms, a value is computed by performing the computation specified in an expression and the value is assigned to a variable. In the fourth form, a pointer, the object on the left side, is made to point to the object or target on the right side. The four forms of the assignment statement are also described in this section.

> **Note:** The Fortran statement syntax in this manual is defined using the same terms used in the Fortran 90 standard. In this section, however, certain terms from the standard have been changed to improve clarity.

Introduction to expressions

7.1

Fortran 90 allows you to define data types, operators for these types, and operators for intrinsic types. These capabilities are provided within the general framework for expressions, which consists of three sets of rules:

- The rules for forming a valid expression
- The rules for interpreting the expression (giving it a meaning)
- The rules for evaluating the expression (how the computation may be carried out)

An expression is formed from operators and operands. There is no change from FORTRAN 77 in the rules for forming expressions, except that a new class of operators has been defined. These are user-defined operators, which are either unary or binary operators. They have the form of a sequence of letters surrounded by periods; `.INVERSE.` and `.PLUS.` are examples of possible user-defined operators.

The formal (BNF) rules for forming expressions imply an order for combining operands with operators. These rules specify that expressions enclosed in parentheses are combined first and that, for example, the multiply operator `*` is combined with its operands before the addition operator + is combined with its operands. This order for operators in the absence of specific parentheses is called the *operator precedence* and is summarized in subsection 7.2.5, page 262. New operators, such as == and >=, have the same precedence and meaning as `.EQ.` and `.GE.`, respectively.

The formation rules for expressions imply that the defined unary operators have highest precedence of all operators, and defined binary operators have the lowest precedence of all operators. When they appear in a context where two or more of these operators of the same precedence are adjacent, the operands are combined with their operators in a left-to-right manner, as is the case for the familiar + and − operators, or in a right-to-left manner for the exponentiation operator (`**`).

Intrinsic operators are generic in the sense that they can operate on operands of different types. For example, the plus operator + operates on operands of type integer as well as real and complex. Intrinsic operators can be extended further by the programmer to operate on operands of types for which there are no intrinsic operations. Similarly, you can use defined unary and defined binary operators to operate on operands of types for which there are no previous definitions. The *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903, describes how any operator can be made generic by the programmer using a generic specifier on an interface block.

The rules for interpretation of an expression are provided by the interpretation of each operator in the expression. When the operator is an intrinsic operator such as +, `*`, or `.NOT.`, and the operands are of intrinsic types allowed for the intrinsic operator, the interpretation is provided by the usual mathematical or symbolic meaning of the operation. Thus, + with two numeric operands means that the two operands are added together. For the user -defined operators, the interpretation is provided by a user-supplied function subprogram with a designation that this subprogram is to be used to define the operation. This aspect is new to Fortran 90. In addition, Fortran 90 allows the intrinsic operator symbols to be extended to cases in which the operands are not of the usual intrinsic types defined by the standard. For example, the + operator can be defined for operands of type `RATIONAL` (a user-defined type) or for operands of type logical with the interpretation provided by a user-supplied function subprogram. The rules for construction of expressions (the syntax rules) are the same for user -defined operators as for intrinsic operators.

The general rule for evaluation of a Fortran expression remains unchanged in Fortran 90. Essentially, the rule states that any method that is mathematically equivalent to that provided by the construction and interpretation rules for the expression is permitted, provided the order of evaluation indicated by explicit parentheses in the expression is followed. Thus, a compiler has a great deal of freedom to rearrange or optimize the computation, provided the rearranged expression has the same mathematical meaning.

Arrays and pointers as objects can appear in expressions and assignment statements. This section describes using arrays and pointers in the following contexts:

- As operands of intrinsic and user-defined operations
- As the variables being assigned in intrinsic assignment statements
- As the variables in pointer assignment statements and masked array assignment statements, new in Fortran 90

Assignment
7.1.1

The result obtained from the evaluation of an expression can be used in many ways. For example, it can be printed or passed to a subprogram. In many cases, however, the value is assigned to a variable and that value can be used later in the program by referencing the variable.

Execution of the assignment statement causes the expression to be evaluated (by performing the computation indicated), and then the value of the expression is assigned to the variable on the left of the equal sign.

Section 7

The following example shows an assignment statement:

```
REAL_AGE = REPORTED_AGE + 3.0
```

`REPORTED_AGE + 3.0` is the expression that indicates how to compute a value, which is assigned to the variable `REAL_AGE`.

The following example involves subscripts:

```
A(I+3) = PI + A(I-3)
```

The value of the subscript expression `I-3` is determined and the value of the `I-3` element of `A` is added to the value of `PI` to produce a sum. Before the result of this expression is assigned, the value of the subscript expression `I+3` is determined and the value of the sum is assigned to the element `I+3` of `A`.

The previous examples are arithmetic; Fortran has expressions of other types, such as logical, character, and derived type. Values of expressions of these other types can be assigned to variables of these other types. As with operators, the programmer can extend the meaning of assignment to types not defined intrinsically and can redefine assignment for two objects of the same derived type. Such assignments are called *defined assignments*. In addition, arrays and pointers each have special forms of assignment statements called *masked array assignment* and *pointer assignment*, respectively . These two assignment statement forms are described later in this section.

Expressions
7.1.2

An assignment statement is only one of the Fortran statements where expressions may occur. Expressions also can appear in subscripts, actual arguments, `IF` statements, `PRINT` statements, `WHERE` statements, declaration statements, and many other statements.

An expression represents a computation that results in a value and can be as simple as a constant or variable. The value of an expression has a type and can have zero, one, or two type parameter values. If the value is of a derived type, it has no type parameter. If it is of an intrinsic type, it has a kind type parameter, and if, in addition, it is of the type character, it has a length type parameter. In addition, the value is a scalar (including a structure) or an array.

A complex value or a structure value is a scalar, even though it can consist of more than one value (for example, a complex value consists of two real values).

Arrays and pointers can be used as operands of intrinsic and defined operators. For intrinsic operators, when an array is an operand, the operation is performed element-wise on the elements of the array. For intrinsic operators, when a pointer is an operand, the value of the target pointed to by (associated with) the pointer is used as the operand. For defined operators, the array or pointer is used in a manner determined by the procedure defining the operation.

As indicated in the introduction to this section, the presentation of expressions is described in terms of the following three basic sets of rules:

- The rules for forming expressions (syntax)
- The rules for interpreting expressions (semantics)
- The rules for evaluating expressions (optimization)

The syntax rules indicate which forms of expressions are valid. The semantics indicate how each expression is to be interpreted. After an expression has been given an interpretation, a compiler can evaluate another completely different expression, provided the expression evaluated is mathematically equivalent to the one written.

To see how this works, consider the expression `2 * A + 2 * B` in the following `PRINT` statement:

```
PRINT *, 2 * A + 2 * B
```

The syntax rules described later in this section indicate that the expression is valid and suggest an order of evaluation. The semantic rules specify the operations to be performed, which in this case, are the multiplication of the values of `A` and `B` by `2` and the addition of the two results. That is, the semantic rules indicate that the expression is to be interpreted as if it were the following:

```
((2 * A) + (2 * B))
```

After the correct interpretation has been determined, the Fortran rules of evaluation allow a different expression to be used to evaluate the expression, provided the different expression is mathematically equivalent to the one written. For example, a processor can first add A and B and then multiply the result by 2, because the following expression is mathematically equivalent to the one written:

```
2 * (A + B)
```

Although alternative evaluations are allowed, three properties should be noted:

- Parentheses must not be violated. Consider the following expression:

  ```
  (2 * A) + (2 * B)
  ```

  This expression must **not** be evaluated as follows:

  ```
  2 * (A + B)
  ```

  This gives the programmer some control over the method of evaluation.

- Integer division is not mathematically equivalent to real division. The value of 3/2 is 1 and so cannot be evaluated as 3*0.5, which is 1.5.

- Mathematically equivalent expressions can produce computationally different results, because of the implementation of arithmetic and rounding on computer systems. For example, the expression X/2.0 could be evaluated as 0.5*X, even though the results may be slightly different. Also, for example, the expression 2 * A + 2 * B could be evaluated as 2*(A+B); when A and B are of type real, the two mathematically equivalent expressions may yield different values because of different rounding errors and different arithmetic exceptions in the two expressions.

Formation of expressions

7.2

An expression is formed from operands, operators, and parentheses. The simplest form of an expression is a constant or a variable.

Expression	Meaning
`3.1416`	A real constant
`.TRUE.`	A logical constant
`X`	A scalar variable
`Y`	An array variable
`Y(K)`	A variable that is an array element of `Y`
`Y(2:10:2)`	A variable that is an array subsection of `Y`
`M%N`	A variable that is a component of a structure `M`
`Y(K)(I:I+3)`	A variable that is a substring of array element `Y(K)`

The values of these simple expressions are the constant value `3.1416`, the constant value `.TRUE.`, the value of the variable `X`, the value of the array `Y`, the value of the array element `Y(K)`, the value of the array subsection `Y(2:10:2)`, the value of the component `N` of structure `M`, and the value of a substring of an array element `Y(K)`, respectively.

Operands
7.2.1

An operand in an expression can be one of the following items:

- A constant or subobject of a constant
- A variable (for example, a scalar, an array, a substring, or a pointer)
- An array constructor
- A structure constructor
- A function reference (returning, for example, a scalar, an array, a character variable, or a pointer)
- Another expression in parentheses

The following examples show operands:

```
A               ! scalar or an array
B(1)            ! array element or function
C(3:5)          ! array section or a substring
(A + COS(X))    ! expression in parentheses
(/ 1.2, 2.41 /)! array constructor
RATIONAL(1,2)   ! structure constructor or function
I_PTR           ! pointer to an integer target
```

Binary and unary operations

7.2.2

There are two forms that operations can take in an expression. One is an operation involving two operands such as multiplying two numbers together . The other is an operation on one operand such as making a number negative. These forms are called *binary* and *unary* operations, respectively .

Table 9 lists the intrinsic operators. You can use function subprograms to define additional operators. User-defined operators are either binary or unary operators.

Table 9. Intrinsic operators and the allowed types of their operands

Operator category	Intrinsic operator	Operand types
Arithmetic	`**`, `*`, `/`, `+`, `−`, unary `+`, unary `−`	Numeric of any combination of numeric types and kind type parameters or Cray pointer. Cray pointers are only allowed with the + or − operators.
Character	//	Character of any length.
Relational	`.EQ.`, `.NE.`, `==`, `/=`	Both of any numeric type and any kind type parameter or Cray pointer, or both of type character with any length type parameter.

Table 9. Intrinsic operators and the allowed types of their operands (continued)

Operator category	Intrinsic operator	Operand types
Relational	`.GT.`, `.GE.`, `.LT.`, `.LE.`, >, >=, <, <=	Both of any numeric type (except complex) and any kind type parameter or Cray pointer, or both of type character with any length type parameter .
Logical	`.NOT.`, `.AND.`, `.OR.`, `.XOR.`, `.EQV.`, `.NEQV.`	Both of type logical with any combination of kind type parameters.
Bitwise masking (Boolean) expressions (CF90 extension)	`.NOT.`, `.AND.`, `.OR.`, `.XOR.`, `.EQV.`, `.NEQV.`	Integer , real, typeless, or Cray pointer.

ANSI/ISO: The Fortran 90 standard does not specify the bitwise masking (Boolean) expressions, nor does it specify the `.XOR.` operator as a logical operator.

A binary operator combines two operands, as in the following:

x_1 *operator* x_2

Examples:

```
A + B
2 * C
```

The examples show an addition between two operands `A` and `B`, and a multiplication of two operands, the constant `2` and the operand `C`.

A unary operation acts on one operand, as in the following:

operator x_1

Examples:

```
- C
.NOT. L
```

The first example results in the value minus `C`. The second example produces a value that is the logical complement of `L`; the operator `.NOT`. is the only intrinsic operator that is a unary operator and is never a binary operator .

Intrinsic and defined operations

7.2.3

Intrinsic operations are those whose definitions are known to the compiler. They are built into Fortran and are always available for use in expressions. Table 9, page 240, lists the operators built into Fortran as specified by the standard.

The relational operator symbols ==, /=, >, >=, <, and <= are new to Fortran with Fortran 90 and are synonyms for the operators `.EQ.`, `.NE.`, `.GT.`, `.GE.`, `.LT.`, and `.LE.`, respectively.

The CF90 compiler allows abbreviations for the logical and masking operators. The abbreviations `.A.`, `.O.`, `.N.`, and `.X.` are synonyms for `.AND.`, `.OR.`, `.NOT.`, and `.XOR.`, respectively. If you define the abbreviated operator for any type, the abbreviated form of the intrinsic operator cannot be used in any scope in which the defined operator is accessible.

> **ANSI/ISO:** The Fortran 90 standard does not specify abbreviations for the logical and masking operators.

In addition to the Fortran operators that are intrinsic (built in), there may be user-defined operators in expressions.

Defined operations are those that you define in the Fortran program and make available to each program unit that uses them. The computation performed by a defined operation is described explicitly in a function that must appear as a subprogram in the Fortran program where it is used. The operator used in a defined operation is called a *defined operator*. In this way, you extend the repertoire of operations so that computations can be expressed in a natural way using operator notation. Function subprograms that define operators are explained in detail in the *CF90 Fortran Language Reference Manual, Volume 2*, publication SR–3903.

A defined operator uses a symbol that is either the symbol for an intrinsic operator or is a new operator symbol. The synonyms described above for the relational operators remain synonyms in all contexts, even when there are defined operators. For example, if the operator < is defined for a new type, say `STRING`, the same definition applies to the operator `.LT.` for the type `STRING`; if the operator `.LT.` is specified as private, the operator < is also private.

A distinction is made between a defined (or new) operator and an extended intrinsic operator. An *extended intrinsic operator* is one that uses the same symbol as an intrinsically defined Fortran operator, like plus + or multiply `*`. It also causes the operations to be combined in the same order as is specified for the intrinsic operator. A *defined operator* is one where the operator symbol is not the same as an intrinsic operator but is new, such as `.INVERSE.`. Defined operators, however, have a fixed precedence; defined unary operators have the highest precedence of all operators and defined binary operators have the lowest precedence of all operators. The precedences of all operators are described in more detail in subsection 7.2.5, page 262.

> **ANSI/ISO:** The masking or Boolean operators and their abbreviations, which are CF90 compiler extensions, can be redefined as defined operators. If you redefine a masking operator, your definition overrides the intrinsic masking operator definition. See table Table 9, page 240, for a list of the operators.

Rules for forming expressions

7.2.4

Expressions are formed by combining operands. Operands can be constants, variables (scalars, array elements, arrays, array sections, structures, structure components, and pointers), array constructors, structure constructors, functions, and parenthesized expressions with intrinsic and defined operators.

The method used to specify the expression formation rules is a collection of syntax rules that determine the forms of expressions. The order of evaluation of the operations in an expression is determined by the usual semantics for the operations, and the syntax rules are designed to be consistent with these semantics. In fact, the order of evaluation defines a precedence order for operators that is summarized in Table 10.

Table 10. The hierarchy of expressions through forms

Term	Definition
expression	[*expression* *defined_operator*] *equivalence_expression*
equivalence_expression	[*equivalence_expression* `.EQV.`] *disjunct_expression* *equivalence_expression* `.NEQV.` *disjunct_expression*
Exclusive OR (Cray Research extension)	[*disjunct_expression* `.XOR.`] *conjunct_expression*
disjunct_expression	[*disjunct_expression* `.OR.`] *conjunct_expression*
conjunct_expression	[*conjunct_expression* `.AND.`] *not_expression*
not_expression	[`.NOT.`] *comparison_expression*
comparison_expression	[*concatenation_expression* *relational_operator*] & *concatenation_expression*
concatenation_expression	[*concatenation_expression* `//`] *summation_expression*
summation_expression	[*summation_expression* +] *multiplication_expression* *summation_expression* − *multiplication_expression* + *multiplication_expression* − *multiplication_expression*
multiplication_expression	[*multiplication_expression* `*`] *exponentiation_expression* *multiplication_expression* / *exponentiation_expression*
exponentiation_expression	*defined_unary_expression* [`**`*exponentiation_expression*]

Table 10. The hierarchy of expressions through forms (continued)

Term	Definition
defined_unary_expression	[*defined_operator*] *primary*
primary	*constant* *constant_subobject* *variable* *array_constructor* *structure_constructor* *function_reference* (*expression*)

The set of syntax rules defines an expression at the highest level in terms of operators and operands, which are themselves expressions. As a result, the formal set of rules is recursive. The basic or lowest level of an expression is a primary, which, for example, can be a variable, a constant, or a function, or recursively an expression enclosed in parentheses. The rules for forming expressions are described from the lowest or most primitive level to the highest or most complex level; that is, the rules are stated from a primary up to an expression.

Primary 7.2.4.1

A primary is defined as follows:

R701	*primary*	**is**	*constant*
		or	*constant_subobject*
		or	*variable*
		or	*array_constructor*
		or	*structure_constructor*
		or	*function_reference*
		or	(*expr*)
R702	*constant_subobject*	**is**	*subobject*

A variable that is a primary must not be an assumed-size array or a section of an assumed-size array name, unless the last subscript position of the array is specified with a scalar subscript or a section subscript in which the upper bound is specified.

The following examples show primaries:

Primary	Meaning
`3.2`	A real constant
`ONE`	A named constant
`'ABCS'(I:I)`	A constant subobject
`A`	A variable (scalar, array, structure, or pointer)
`B(:,1:N)`	An assumed-size array with an upper bound in the last dimension
`C(I)`	An array element
`CH(I:J)`	A substring
`(/ 1, J, 7 /)`	An array constructor
`RATIONAL(I, J)`	A structure constructor
`FCN(A)`	A function reference
`(A * B)`	A parenthesized expression

In the previous examples, `ONE` is a named constant if it has the `PARAMETER` attribute or appears in a `PARAMETER` statement. `'ABCS'(I:I)` is a constant subobject even though `I` may be a variable because its parent (`'ABCS'`) is a constant; the reference `'ABCS'(I:I)` is a constant subobject because it cannot be defined like a variable can be defined. `RATIONAL` is a derived type and `FCN` is a user-defined function.

When an array variable is a primary, the whole array is used, except in a masked assignment statement. In a masked assignment statement, only that part of the array specified by the mask is used.

When a pointer is a primary, the target associated with (pointed to by) the pointer is used, except possibly when the pointer is an actual argument of a procedure, or is an operand of a defined operation or a defined assignment. Whether the pointer or the target is used in these exceptional cases is determined by the procedure invoked by the reference.

Recall that an assumed-size array is a dummy argument whose shape is not completely specified in the subprogram in that the extent in the last dimension is determined by its corresponding actual argument. The implementation model is that the extent in the last dimension is never known to the subprogram but is specified by the use of a subscript, section subscript, or vector subscript expression that defines an upper bound in the last dimension. Unless the extent is specified in this way, such an object must not be used as a primary in an expression. On the other hand, if a subscript, section subscript with an extent for the upper bound, or a vector subscript is specified for the last dimension, the array value has a well-defined shape and hence can be used as a primary in any expression. For example, if `A` is declared as `REAL A(3,*)`, array `A(:,3)` has a well-defined shape and can be used as a primary in an expression.

Expressions can be used as actual arguments in procedure references (function references or subroutine calls). Because actual arguments can be expressions involving operations, actual arguments must not contain assumed-size arrays, unless their shape is well-defined, as described above. An actual argument, however, can be just a variable, which then allows the actual argument to be the name of an assumed-size array. This implies that such actual arguments can be assumed-size arrays, unless the procedure requires the shape of the argument to be specified by the actual argument. Most of the intrinsic procedures that allow array arguments require the shape to be specified for the actual array arguments, and therefore assumed-size arrays cannot be used as actual arguments for most intrinsic functions. The exceptions are all references to the intrinsic function `LBOUND`, and certain references to the intrinsic functions `UBOUND` and `SIZE`.

7.2.4.2 Defined-unary expression

Defined-unary expressions have the highest operator precedence. A *defined-unary expression* is a defined operator followed by a primary. These are defined as follows:

R703	*defined_unary_expr*	**is**	[*defined_operator*] *primary*
R704	*defined_operator*	**is**	. *letter* [*letter*] ... .

A defined operator must not contain more than 31 letters.

A defined operator must not be the same as any intrinsic operator (`.NOT.`, `.AND.`, `.OR.`, `.EQV.`, `.NEQV.`, `.EQ.`, `.NE.`, `.GT.`, `.GE.`, `.LT.`, or `.LE.`) or any logical literal constant (`.FALSE.` or `.TRUE.`).

A defined operator can be the same as one of the CF90 compiler masking or Boolean operators. The corresponding CF90 compiler operator loses its intrinsic properties. Note that the abbreviations `.T.`, `.F.`, `.A.`, `.O.`, `.N.`, and `.X.` are synonyms for `.TRUE.`, `.FALSE.`, `.AND.`, `.OR.`, `.NOT.`, and `.XOR.`, respectively. If you define the abbreviated operator for any type, the abbreviated form of the intrinsic operator also cannot be used in any scope in which the defined operator is accessible, and the redefined abbreviated logical constants can no longer be used as logical constants.

The following examples show defined-unary expressions:

Expression	Meaning
`.INVERSE. B`	A defined-unary expression (where `.INVERSE.` is a defined operator)
`A`	A primary is also a defined-unary expression

Exponentiation expression
7.2.4.3

An *exponentiation expression* is an expression in which the operator is the exponentiation operator `**`. This is defined as follows:

R705	*exponentiation_expr*	**is**	*defined_unary_expr* [`**` *exponentiation_expr*]

Note that the definition is right recursive (that is, the defined term appears to the right of the operator `**`) which indicates that the precedence of the `**` operator in contexts of equal precedence is right-to-left. Thus, the interpretation of the expression `A ** B ** C` is `A ** ( B ** C )`.

The following examples show exponentiation expressions:

Expression	Meaning
`A ** B`	An exponentiation expression
`A ** B ** C`	An exponentiation expression with right-to-left precedence
`.INVERSE. B`	A defined-unary expression is also an exponentiation expression
`A`	A primary is also an exponentiation expression

Multiplication expression
7.2.4.4

A multiplication expression is an expression in which the operator is either `*` or `/`. It is defined as follows:

R706	*multiplication_expr*	**is**	[*multiplication_expr* `*`] *exponentiation_expr*
		or	[*multiplication_expr* `/`] *exponentiation_expr*

Note that the definition is left recursive (that is, the defined term appears to the left of the operator `*` or `/`) which indicates that the precedence of the `*` and `/` operators in contexts of equal precedence is left-to-right. Thus, the interpretation of the expression `A * B * C` is `(A * B) * C`, or `A / B * C` is `(A / B) * C`. This left-to-right precedence rule applies to the remaining binary operators except the relational operators.

The following examples show multiplication expressions:

Expression	Meaning
`A * B`	A multiplication expression
`A * B * C`	A multiplication expression with left-to-right precedence
`A / B`	A multiplication expression
`A / B / C`	A multiplication expression with left-to-right precedence
`A ** B`	An exponentiation expression is also a multiplication expression
`.INVERSE. B`	A defined-unary expression is also a multiplication expression
`A`	A primary is also a multiplication expression

Summation expression
7.2.4.5

A *summation expression* is an expression in which the operator is either + or −. It is defined as follows:

R707	*summation_expr*	**is**	[[*summation_expr*] +] *multiplication_expr*
		or	[[*summation_expr*] −] *multiplication_expr*

The following examples show summation expressions:

Expression	Meaning
`A + B`	A summation expression
`A + B - C`	A summation expression with left-to-right precedence
`- A - B - C`	A summation expression with left-to-right precedence
`+ A`	A summation expression using unary +
`- A`	A summation expression using unary −
`A * B`	A multiplication expression is also a summation expression
`A ** B`	An exponentiation expression is also a summation expression
`.INVERSE. B`	A defined-unary expression is also a summation expression
`A`	A primary is also a summation expression

Concatenation expression 7.2.4.6

A *concatenation expression* is an expression in which the operator is `//`. It is defined as follows:

R711	*concatenation_expr*	**is**	[*concatenation_expr* //] *summation_expr*

The following examples show concatenation expressions:

Expression	Meaning
`A // B`	A concatenation expression
`A // B // C`	A concatenation expression with left-to-right precedence

Section 7

Expression	Meaning
`A - B`	A summation expression is also a concatenation expression
`- A`	A summation expression is also a concatenation expression
`A * B`	A multiplication expression is also a concatenation expression
`A ** B`	An exponentiation expression is also a concatenation expression
`.INVERSE. B`	A defined-unary expression is also a concatenation expression
`A`	A primary is also a concatenation expression

7.2.4.7 Comparison expression

A *comparison expression* is an expression in which the operator is a relational operator. It is defined as follows:

R713	*comparison_expr*	**is**	[*concatenation_expr* *rel_op*] *concatenation_expr*
R714	*rel_op*	**is**	`.EQ.`
		or	`.NE.`
		or	`.LT.`
		or	`.LE.`
		or	`.GT.`
		or	`.GE.`
		or	`==`
		or	`/=`
		or	`<`
		or	`<=`
		or	`>`
		or	`>=`

The operators ==, /=, <, <=, >, and >= are synonyms in all contexts for the operators `.EQ.`, `.NE.`, `.LT.`, `.LE.`, `.GT.`, and `.GE.`, respectively.

Note that the definition of a comparison expression is not recursive, and therefore comparison expressions cannot contain relational operators in contexts of equal precedence.

The following examples show comparison expressions:

Expression	Meaning
`A .EQ. B`	A comparison expression
`A < B`	A comparison expression
`A // B`	A concatenation expression is also a comparison expression
`A - B`	A summation expression is also a comparison expression
`- A`	A summation expression is also a comparison expression
`A * B`	A multiplication expression is also a comparison expression
`A ** B`	An exponentiation expression is also a comparison expression
`.INVERSE. B`	A defined-unary expression is also a comparison expression
`A`	A primary is also a comparison expression

7.2.4.8 Not expression

A *not expression* is an expression in which the operator is `.NOT.`. It is defined as follows:

R715	*not_expr*	**is**	[`.NOT.`] *comparison_expr*

Note that the definition of a not expression is not recursive, and therefore not expressions cannot contain adjacent `.NOT.` operators.

The following examples show not expressions:

Expression	Meaning
`.NOT. A`	A not expression
`A .EQ. B`	A comparison expression is also a not expression
`A // B`	A concatenation expression is also a not expression
`A - B`	A summation expression is also a not expression
`- A`	A summation expression is also a not expression
`A * B`	A multiplication expression is also a not expression
`A ** B`	An exponentiation expression is also a not expression
`.INVERSE. B`	A defined-unary expression is also a not expression
`A`	A primary is also a not expression

Conjunct expression
7.2.4.9

A *conjunct expression* is an expression in which the operator is `.AND.`. It is defined as follows:

R716	*conjunct_expr*	**is**	[*conjunct_expr* `.AND.`] *not_expr*

Note that the definition of a conjunct expression is left recursive, and therefore the precedence of the `.AND.` operator in contexts of equal precedence is left-to-right. Thus, the interpretation of the expression `A .AND. B .AND. C` is `(A .AND. B) .AND. C`.

The following examples show conjunct expressions:

Expression	Meaning
`A .AND. B`	A conjunct expression
`A .AND. B .AND. C`	A conjunct expression with left-to-right precedence

Expression	Meaning
`.NOT. A`	A not expression is also a conjunct expression
`A .EQ. B`	A comparison expression is also a conjunct expression
`A // B`	A concatenation expression is also a conjunct expression
`A - B`	A summation expression is also a conjunct expression
`- A`	A summation expression is also a conjunct expression
`A * B`	A multiplication expression is also a conjunct expression
`A ** B`	An exponentiation expression is also a conjunct expression
`.INVERSE. B`	A defined-unary expression is also a conjunct expression
`A`	A primary is also a conjunct expression

Inclusive disjunct expression 7.2.4.10

An inclusive disjunct expression is an expression in which the operator is `.OR.`. It is defined as follows:

R717	*inclusive_disjunct_expr*	**is**	[*inclusive_disjunct_expr* `.OR.`] *conjunct_expr*

Note that the definition of an inclusive disjunct expression is left recursive, and therefore the precedence of the `.OR.` operator in contexts of equal precedence is left-to-right. Thus, the interpretation of the expression `A .OR. B .OR. C` is `(A .OR. B) .OR. C`.

The following examples show inclusive disjunct expressions:

Expression	Meaning
`A .OR. B`	An inclusive disjunct expression
`A .OR. B .OR. C`	An inclusive disjunct expression with left-to-right precedence
`A .AND. B`	A conjunct expression is also an inclusive disjunct expression
`.NOT. A`	A not expression is also an inclusive disjunct expression
`A .EQ. B`	A comparison expression is also an inclusive disjunct expression
`A // B`	A concatenation expression is also an inclusive disjunct expression
`A - B`	A summation expression is also an inclusive disjunct expression
`- A`	A summation expression is also an inclusive disjunct expression
`A * B`	A multiplication expression is also an inclusive disjunct expression
`A ** B`	An exponentiation expression is also an inclusive disjunct expression
`.INVERSE. B`	A defined-unary expression is also an inclusive disjunct expression
`A`	A primary is also an inclusive disjunct expression

Equivalence expressions and exclusive disjunct expressions 7.2.4.11

An *equivalence expression* is an expression in which the operator is either `.EQV.` or `.NEQV.`. It is defined as follows:

R718	*equivalence_expr*	**is**	[*equivalence_expr* `.EQV.`] *inclusive_disjunct_expr*
		or	[*equivalence_expr* `.NEQV.`] *inclusive_disjunct_expr*

An *exclusive disjunct expression* is an expression in which the operator is `.XOR.`. It is defined as follows:

CRI	*exclusive_disjunct_expr*	**is**	[*exclusive_disjunct_expr* `.XOR.`] ≡ ≡ *inclusive_disjunct_expr*

ANSI/ISO: The Fortran 90 standard does not specify the `.XOR.` operator .

Note the following:

- In the following discussion, *equivalence expression* means either *equivalence expression* or *exclusive disjunct expression*.
- The definition of an equivalence expression is left recursive, and therefore the precedence of the `.EQV.` or `.NEQV.` operators in contexts of equal precedence is left-to-right. Thus, the interpretation of the expression `A .EQV. B .NEQV. C` is `(A .EQV. B) .NEQV. C`.

The following examples show equivalence expressions:

Expression	Meaning
`A .EQV. B`	An equivalence expression
`A .XOR. B`	An equivalence exclusive disjunct expression
`A .NEQV. B .XOR. C`	An equivalence expression with left-to-right precedence

Section 7

Expression	Meaning
`A .OR. B`	An inclusive disjunct expression is also an equivalence expression
`A .AND. B`	A conjunct expression is also an equivalence expression
`.NOT. A`	A not expression is also an equivalence expression
`A .EQ. B`	A comparison expression is also an equivalence expression
`A // B`	A concatenation expression is also an equivalence expression
`A - B`	A summation expression is also an equivalence expression
`- A`	A summation expression is also an equivalence expression
`A * B`	A multiplication expression is also an equivalence expression
`A ** B`	An exponentiation expression is also an equivalence expression
`.INVERSE. B`	A defined-unary expression is also an equivalence expression
`A`	A primary is also an equivalence expression

Expression 7.2.4.12

The most general form of an expression is defined as follows:

R723	*expr*	**is**	[*expr* *defined_binary_op*] *equivalence_expr*
R724	*defined_binary_op*	**is**	**.** *letter* [*letter*] ... **.**

Note that the definition of an expression is left recursive, and therefore the precedence of the binary defined operator in contexts of equal precedence is left-to-right. The interpretation of the expression `A .PLUS. B .MINUS. C` is thus `(A .PLUS. B) .MINUS. C` (where `.MINUS.` is a defined operator).

The following examples show expressions:

Expression	Meaning
`A .PLUS. B`	An expression (where `.PLUS.` is a defined operator)
`A .CROSS. B .CROSS. C`	An expression with left-to-right precedence (where `.CROSS.` is a defined operator)
`A .EQV. B`	An equivalence expression is also an expression
`A .OR. B`	An inclusive disjunct expression is also an expression
`A .AND. B`	A conjunct expression is also an expression
`.NOT. A`	A not expression is also an expression
`A .EQ. B`	A comparison expression is also an expression
`A // B`	A concatenation expression is also an expression
`A - B`	A summation expression is also an expression
`- A`	A summation expression is also an expression
`A * B`	A multiplication expression is also an expression
`A ** B`	An exponentiation expression is also an expression
`.INVERSE. B`	A defined-unary expression is also an expression
`A`	A primary is also an expression

Summary of the forms and hierarchy for expressions 7.2.4.13

The previous subsections have described in detail the sorts of expressions that can be formed. These expressions form a hierarchy that can best be illustrated by a figure. Figure 13, page 261, describes the hierarchy by placing the simplest form of an expression, a variable, at the center of a set of nested rectangles. The more general forms of an expression are the enclosing rectangles, from a primary to an exponentiation expression, to a summation expression, and finally to a general expression using a defined binary operator `.CROSS.`. Figure 13 demonstrates that an expression is all of these special case forms, including the simplest form, a primary.

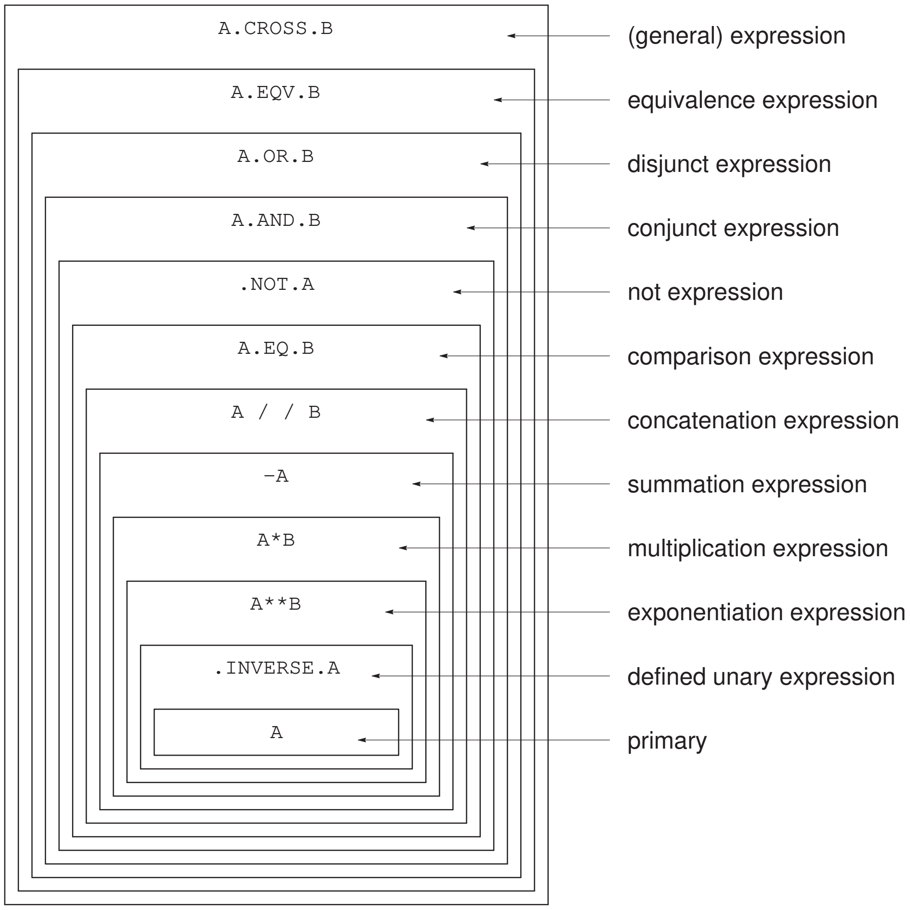

Figure 13. The hierarchy of expressions by examples

Table 10, page 244, illustrates the relationship between the different sorts of expressions by summarizing the definitional forms in one table. The simplest form of an expression is at the bottom and is the primary, as in Figure 13. The next, more general, form is second from the bottom and is the defined unary expression; it uses the primary in its definition. At the top of the table is the most general form of an expression.

Precedence of operators
7.2.5

Table 11 summarizes the relative precedence of operators, including the precedence when operators of equal precedence are adjacent. An entry `N/A` in the rightmost column indicates that the operator cannot appear in such contexts. The leftmost column classifies the operators as defined, numeric, character , relational, and logical operators. Note that these operators are not intrinsic operators unless the types of the operands are those specified in Table 12, page 264.

Table 11. Categories of operations and relative precedences

Category of operator	Operator	Precedence	In context of equal precedence
Defined	Unary defined-operator	Highest	N/A
Numeric	**	.	Right-to-left
Numeric	* or /	.	Left-to-right
Numeric	Unary + or −	.	N/A
Numeric	Binary + or −	.	Left-to-right
Character	//	.	Left-to-right
Relational	`.EQ.`, `.NE.`, `.LT.`, `.LE.`, `.GT.`, `.GE.`, ==, /=, < <=, >, >=	.	N/A
Logical or Boolean	`.NOT.`	.	N/A
Logical or Boolean	`.AND.`	.	Left-to-right
Logical or Boolean	`.OR.`	.	Left-to-right
Logical	`.EQV.` or `.NEQV.`	.	Left-to-right

Table 11. Categories of operations and relative precedences (continued)

Category of operator	Operator		Precedence	In context of equal precedence
Logical or Boolean		.XOR.	.	Left-to-right
Defined	Binary	defined-operator	Lowest	Left-to-right

For example, consider the following expression:

```
A .AND. B .AND. C .OR. D
```

Table 10 indicates that the .AND. operator is of higher precedence than the .OR. operator, and the .AND. operators are combined left-to-right when in contexts of equal precedence; thus, A and B are combined by the .AND. operator , the result A .AND. B is combined with C using the .AND. operator , and that result is combined with D using the .OR. operator. Thus, this expression is interpreted the same way as the following fully parenthesized expression:

```
(((A .AND. B) .AND. C) .OR. D)
```

Notice that the defined operators have fixed precedences; defined unary operators have the highest precedence of all operators and are all of equal precedence; defined binary operators have the lowest precedence, are all of equal precedence, and are combined left-to-right when in contexts of equal precedence. Both kinds of defined operators may have multiple definitions in the program unit and therefore may be generic just as intrinsic operators and intrinsic procedures are generic.

As a consequence of the expression formation rules, unary operators cannot appear in a context of equal precedence; the precedence must be specified by parentheses. There is thus no left-to-right or right-to-left rule for any unary operators. Similarly, the relational operators cannot appear in a context of equal precedence; consequently, there is no left-to-right or right-to-left rule for the relational operators. Use of some of the operators as Boolean or masking operators is a CF90 compiler extension.

ANSI/ISO: The Fortran 90 standard does not specify the use of operators as Boolean or masking operators.

Intrinsic operations
7.2.6

Intrinsic operations are those known to the processor. For an operation to be intrinsic, an intrinsic operator symbol must be used, and the operands must be of the intrinsic types specified in Table 12.

Note: In the following table, the symbols I, R, Z, C, L, B, and P stand for the types integer, real, complex, character, logical, Boolean, and Cray pointer , respectively. Where more than one type for x_2 is given, the type of the result of the operation is given in the same relative position in the next column. Boolean and Cray pointer types are CF90 compiler extensions.

Table 12. Operand types and results for intrinsic operations

Intrinsic operator	Type of x_1	Type of x_2	Type of result
Unary +, −		I, R, Z, B, P	I, R, Z, I, P
Binary +, −, *, /, **	I	I, R, Z, B, P	I, R, Z, B, P
	R	I, R, Z, B	R, R, Z, R
	Z	I, R, Z	Z, Z, Z
	B	I, R, B, P	I, R, B, P
	P (For Cray pointer , only + and − are allowed.)	I, B, P	P, P, P
//	C	C	C

Table 12. Operand types and results for intrinsic operations (continued)

Intrinsic operator	Type of x_1	Type of x_2	Type of result
.EQ., ==, .NE., /=	I	I, R, Z, B, P	L, L, L, L, L
	R	I, R, Z, B, P	L, L, L, L, L
	Z	I, R, Z, B, P	L, L, L, L, L
	B	I, R, Z, B, P	L, L, L, L, L
	P	I, R, Z, B, P	L, L, L, L, L
	C	C	L
.GT., >, .GE., >=, .LT., <, .LE., <=	I	I, R, B, P	L, L, L, L
	R	I, R, B	L, L, L
	C	C	L
	P	I, P	L, L
.NOT.		L	L
		I, R, B	B
.AND., .OR., .EQV., .NEQV., .XOR.	L	L	L
	I, R, B	I, R, B	B

ANSI/ISO: The Fortran 90 standard does not specify the use of type Boolean or the use of type Boolean in masking expressions.

The intrinsic operations are either binary or unary. The binary operations use the binary intrinsic operator symbols +, −, *, /, **, //, .EQ., .NE., .LT., .GT., .LE., .GE. (and their synonyms ==, /=, <>, <, and >=), .AND., .OR., .XOR., .EQV., and .NEQV.. The unary operations use the unary intrinsic operator symbols +, −, and .NOT..

Note that the intrinsic operators `.AND.`, `.OR.`, `.NOT.`, and `.XOR.` can be abbreviated as `.A.`, `.O.`, `.N.`, or `.X.`. If a user-defined operator with the same name as the abbreviated name is accessible in a scope, the abbreviated forms of these operators may not be used as synonyms for `.AND.`, `.OR.`, `.NOT.`, or `.XOR.`.

The intrinsic operations are divided into five classes with different rules and restrictions for the types of the operands. The five classes are numeric intrinsic, character intrinsic, logical intrinsic, relational intrinsic operations, and bitwise masking expressions.

The numeric intrinsic operations use the intrinsic operators `+`, `−`, `*`, `/`, and `**`. The operands can be of any numeric type and with any kind type parameters. The result of the operation is of a type specified by Table 12 and has type parameters as specified in subsection 7.2.8.2, page 271.

For example, the following expressions, in which `I`, `R`, `D`, and `Z` are declared to be of types integer, real, double-precision real, and complex, have the types and type parameters of the variables `R`, `I`, `D`, and `Z`, respectively:

```
I + R
I * I
I - D
I / Z
```

There is only one character intrinsic operation; it uses the intrinsic operator `//`. The result of a character intrinsic operation is type character.

The logical intrinsic operations use the intrinsic operators `.AND.`, `.OR.`, `.NOT.`, `.EQV.`, and `.NEQV.`, respectively. The result of a logical intrinsic operation is type logical and has type parameters as specified in subsection 7.2.8.2, page 271.

The relational intrinsic operations use the intrinsic operators `.EQ.`, `.NE.`, `.GT.`, `.GE.`, `.LT.`, `.LE.`, or their symbolic synonyms. A relational intrinsic operation is a numeric relational intrinsic operation if its operands are of type integer, real, or complex. A relational intrinsic operation is a character relational intrinsic operation if its operands are of type character. The result of either kind of relational operation is type logical and has type parameters as specified in subsection 7.2.8.2, page 271.

The operators `.NOT.`, `.AND.`, `.OR.`, `.EQV.`, and `.XOR.` can also be used in the CF90 compiler extension bitwise masking expressions. The result is Boolean (typeless) and has no kind type parameters.

Defined operations

7.2.7

A *defined operation* is any nonintrinsic operation that is interpreted and evaluated by a function subprogram specified by an interface block with a generic specifier of the following form:

`OPERATOR` (*defined_operator*)

A defined operation uses either a defined operator or an intrinsic operator symbol, and it is either unary or binary. Its forms (R703, R723) are as follows:

intrinsic_unary_op x_2

defined_operator x_2

x_1 *intrinsic_binary_op* x_2

x_1 *defined_operator* x_2

The terms *intrinsic_unary_op* and *intrinsic_binary_op* include all intrinsically defined operators; these terms are not specifically defined in any syntax rules.

Section 7

x_1 and x_2 are operands. When an intrinsic operator symbol is used, the type of x_2 (for a unary operator) and types of x_1 and x_2 (for a binary operator) must not be the same as the types of the operands specified in Table 12, page 264, for the particular intrinsic operator symbol. Thus, you cannot redefine intrinsic operations on intrinsic types.

When a defined operation uses an intrinsic operator symbol, the generic properties of that operator are extended to the new types specified by the interface block. When a defined operation uses a defined operator, the defined operation is called an *extension operation,* and the operator is called an *extension operator* . An extension operator can have generic properties by specifying more than one function subprogram in an interface block with a generic specifier of the form `OPERATOR (`*defined_operator* `)`.

Data type, type parameters, and shape of an expression

7.2.8

The data type, type parameters, and shape of a complete expression are determined by the data type, type parameters, and shape of each constant, variable, constructor, and function reference appearing in the expression. The determination is inside-out in the sense that the properties are determined first for the primaries. These properties are then determined repeatedly for the operations in precedence order, resulting eventually in the properties for the expression.

For example, consider the expression `A + B * C`, where `A`, `B`, and `C` are of numeric type. First, the data types, type parameter values, and shapes of the three variables `A`, `B`, and `C` are determined. Because `*` has a higher precedence than `+`, the operation `B * C` is performed first. The type, type parameters, and shape of the expression `B * C` are determined next, and then these properties for the entire expression are determined from those of `A` and `B * C`.

Data type and type parameters of a primary

7.2.8.1

The type, type parameters, and shape of a primary that is a nonpointer variable or constant are straightforward because these properties are determined by specification statements for the variable or named constant, or by the form of the constant. For example, if `A` is a variable, its declaration in a specification statement such as the following determines it as an explicit-shaped array of type real with a default kind parameter:

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

For a constant such as the following, the form of the constant indicates that it is a scalar constant of type complex and of default kind:

```
(1.3, 2.9)
```

For a pointer variable, the type, type parameters, and rank are determined by the declaration of the pointer variable. However, if the pointer is of deferred shape, the shape (in particular, the extents in each dimension) is determined by the target of the pointer. Consider the following declarations and assume that pointer `A` is associated with the target `B`:

```
REAL, POINTER :: A(:, :)
REAL, TARGET  :: B(10, 20)
```

The shape of `A` is `(10, 20)`.

The type and type parameters of an array constructor are determined by the contents of the constructor. Unless the element is of type Boolean (typeless), its type and type parameters are those of any element of the constructor because they must all be of the same type and type parameters. If the element is of type Boolean, the type and kind type of the array constructor are the same as the default integer type. Therefore, the type and type parameters of the following array constructor are integer and kind value `1`:

```
(/ 1_1, 123_1, -10_1 /)
```

Its shape is always of rank one and of size equal to the number of elements.

The type of a structure constructor is the derived type used as the name of the constructor. A structure has no type parameters. So, the type of the following structure constructor is the derived type `PERSON`. (See subsection 4.6, page 125, for the type definition `PERSON`.)

```
PERSON(56, 'Father')
```

A structure constructor is always a scalar.

The type, type parameters, and shape of a function are determined by one of the following:

- An implicit type declaration for the function within the program unit referencing the function
- An explicit type declaration for the function within the program unit referencing the function (just like a variable)
- An explicit interface to the function. (When the interface is not explicit, the function is either an external function or a statement function.)

If the interface is explicit, the type, type parameter, and shape are determined by one of the following:

- The type and other specification statements for the function in an interface block within the program unit referencing the function
- The type and other specification statements for the internal or module procedure specifying the function
- The description of the particular intrinsic function being referenced

Note, however, that because intrinsic functions and functions with interface blocks can be generic, these properties are determined by the type, type parameters, and shapes of the actual arguments of the particular function reference.

For example, consider the following statements as part of the program unit specifying an internal function `FCN`:

```
REAL FUNCTION FCN(X)
DIMENSION FCN(10, 15)
```

A reference to `FCN(3.3)` is of type default real with shape `(10,15)`. As a second example, consider the following:

```
REAL(SINGLE)  X(10, 10, 10)
    . . .
    . . .  SIN(X)   . . .
```

The interface to `SIN` is specified by the definition of the sine intrinsic function. In this case, the function reference `SIN (X)` is of type real with kind parameter value `SINGLE` and of shape `(10,10,10)`.

The interface is implicit if the function is external (and no interface block is provided) or is a statement function. In these cases, the shape is always that of a scalar, and the type and type parameters are determined by the implicit type declaration rules in effect, or by an explicit type declaration for the function name. In the following example, `FCN(X)` is a scalar of type integer with kind type parameter value `SHORT`:

```
IMPLICIT INTEGER(SHORT) (A-F)
   . . .
   . . .  FCN(X)    . . .
```

The one case for variables and functions that is not straightforward is the determination of the shape of a variable when it is of deferred shape or of assumed shape. For a deferred-shape array, the rank is known from the declaration but the size of each dimension is determined as the result of executing an `ALLOCATE` statement or a pointer assignment statement. For an assumed-shape array, the rank is also known from the declaration but the size is determined by information passed into the subprogram. In the case of pointers, the shape of the object is that of the target associated with (pointed to by) the pointer. The shape of deferred-shape and assumed-shape arrays thus cannot be determined in general until execution time.

7.2.8.2 Type and type parameters of the result of an operation

The type of the result of an intrinsic operation is determined by the type of the operands and the intrinsic operation and is specified by Table 12, page 264.

For nonnumeric operations, the type parameters of the result of an operation are determined as follows:

- For the relational intrinsic operations, the kind type parameter is that for the default logical type.
- For logical intrinsic operations, the result kind type parameter depends on that of the operands. If the operands have the same kind type parameter (that is the same *value* for `KIND=`*value*), the kind type parameter is that of the operands. If logical operands have different kind type parameter values, the value of the result is that of the greater kind type parameter value.

- For the character intrinsic operation (//), the operands must have the same kind type parameter, so the result has that kind type parameter.
- The character length type parameter value for the result is the sum of the length type parameters of the operands.

For numeric intrinsic operations, the kind type parameter value of the result is determined as follows:

- For unary operations, the kind type parameter value of the result is that of the operand.
- Floating-point operands are those of type real or type complex. If an operation includes a floating-point operand and an integer, the result type and kind type parameter value is determined as follows:
  - The result type is that of the floating-point operand.
  - The result kind type parameter value is the same as the kind type parameter value of the floating-point operand.

  See Table 12, page 264, for more information.
- If an operation includes operands of type complex and type real, the result type and kind type parameter value is determined as follows:
  - The result type is type complex.
  - The result kind type parameter value is that of the greater value. That is, the kind type parameter value of each operand is examined, and whichever kind type parameter value is greater is assigned to the result.

  See Table 12, page 264, for more information.
- For binary operations, if the operands are of the same type and kind type parameters, the kind type parameter value of the result is the kind type parameter of the operands.

- For binary operations, if the operands are both of type integer but with different kind type parameters, the kind type parameter value of the result is the kind type parameter of the operand with the larger decimal exponent range. If the decimal exponent ranges of the two kinds are the same, the kind type parameter value of the result is the same as each operand.
- For binary operations, if the operands are both of type real or complex but with different kind type parameters, the kind type parameter of the result is the kind type parameter of the operand with the larger decimal precision. If the decimal precisions are the same, the kind type parameter value is that of the operands.

For numeric intrinsic operations, an easy way to remember the result type and type parameter rules is to consider that the numeric types (integer, real, complex, and CF90 compiler Boolean) are ordered by the increasing generality of numbers. Integers are contained in the set of real numbers and real numbers are contained in the set of complex numbers. Within the integer type, the kinds are ordered by increasing decimal exponent ranges. Within the real and complex types, the kinds for each type are ordered by increasing decimal precision.

Using this model, the result type of a numeric intrinsic operation is the same type as the operand of the greater generality. For the result type parameter , the rule is complicated: if one or both of the operands is of type real or complex, the type parameter is that of the set of numbers of the more general type described above and with a precision as large as the precision of the operands; if both are of type integer, the result type parameter is of a set of numbers that has a range as large as the range of the operands.

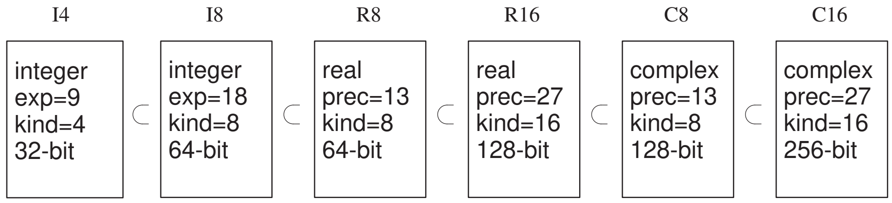

Figure 14. Example ordering of numeric types on Cray PVP systems

To illustrate this ordering, consider Figure 14, which shows ordering on Cray PVP systems. For integers, `KIND=4` is 32-bit format and `KIND=8` is a 64-bit format, with decimal exponent ranges 9 and 18. For the two kinds of reals, `KIND=8` is a 64-bit format and `KIND=16` is a 128-bit format, with decimal precisions 13 and 27. For the two kinds of complex, `KIND=8` is a 128-bit format and `KIND=16` is a 256-bit format, with decimal precisions 13 and 27. Let variables of the 6 numeric types be `I4`, `I8`, `R8`, `R16`, `C8`, and `C16`, where the letter designates the type and the digits designate the kind type parameter. Using this ordering, Table 13 shows the type and type parameters of some simple expressions.

Table 13. Type and type parameters of some simple expressions

Expressions	Type and type parameters are the same as the variable
`I4 + R8`	`R8`
`I8 * C16`	`C16`
`C8 / C16`	`C16`
`I4 - I8`	`I8`
`I4 ** C8`	`C8`
`R16 + C8`	`C16`
`C8 ** I4`	`C8`
`I8 - R8`	`R8`

The type and type parameter values of a defined operation are determined by the interface block for the referenced operation and are the type and type parameters of the name of the function specified by the interface block. Note that the operator can be generic and therefore the type and type parameters can be determined by the operands. For example, consider the following interface:

```
INTERFACE OPERATOR (.PLUS.)

   TYPE(SET) FCN_SET_PLUS(X, Y)
      USE DEFINITIONS
      TYPE (SET)  X, Y
      INTENT(IN) X, Y
   END FUNCTION FCN_SET_PLUS

   TYPE(RATIONAL) FCN_RAT_PLUS(X, Y)
      USE DEFINITIONS
      TYPE(RATIONAL)  X, Y
      INTENT(IN) X, Y
   END FUNCTION FCN_RAT_PLUS

END INTERFACE
```

The operation `A .PLUS. B`, where `A` and `B` are of type `RATIONAL`, is an expression of type `RATIONAL` with no type parameters. The operation `C .PLUS. D`, where `C` and `D` are of type `SET` is an expression of type `SET` with no type parameters.

Shape of an expression 7.2.8.3

The shape of an expression is determined by the shape of each operand in the expression in the same recursive manner as for the type and type parameters for an expression. That is, the shape of an expression is the shape of the result of the last operation determined by the interpretation of the expression.

However , the shape rules are simplified considerably by the requirement that the operands of binary intrinsic operations must be in shape conformance; two operands are in *shape conformance* if both are arrays of the same shape, or one or both operands are scalars. The operands of a defined operation have no such requirement but must match the shape of the corresponding dummy arguments of the defining function.

For primaries that are constants, variables, constructors, or functions, the shape is that of the constant, variable, constructor, or function name. Recall that structure constructors are always scalar, and array constructors are always rank-one arrays of size equal to the number of elements in the constructor . For unary intrinsic operations, the shape of the result is that of the operand. For binary intrinsic operations, the shape is that of the array operand if there is one and is scalar otherwise. For defined operations, the shape is that of the function name specifying the operation.

For example, consider the intrinsic operation `A + B` where `A` and `B` are of type default integer and default real respectively; assume `A` is a scalar and `B` is an array of shape `(3, 5)`. Then, the result is of type default real with shape `(3, 5)`.

The extents of an expression
7.2.8.4

For most contexts, the lower and upper bounds of an array expression are not needed; only the sizes of each dimension are needed to satisfy array conformance requirements for expressions. The bounds of an array expression when it is the `ARRAY` argument (first positional argument) of the `LBOUND` and `UBOUND` intrinsic functions are needed, however.

The functions `LBOUND` and `UBOUND` have two keyword arguments `ARRAY` and `DIM`; `ARRAY` is an array expression and `DIM`, which is optional, is an integer . If the `DIM` argument is present, `LBOUND` and `UBOUND` return the lower and upper bounds, respectively, of the dimension specified by the `DIM` argument. If `DIM` is absent, they return a rank-one array of the lower and upper bounds, respectively, of all dimensions of the `ARRAY` argument. As described below, these functions distinguish the special cases when the array argument is a name or structure component with no section subscript list from the general case when the array argument is a more general expression. Note that if `A` is a structure with an array component `B`, `A%B` is treated as if it were an array name and not an expression.

When the `ARRAY` argument is an array expression that is not a name or a structure component, the function `LBOUND` returns `1` if the `DIM` argument is specified and returns a rank-one array of 1s if the `DIM` argument is absent. For the same conditions, the function `UBOUND` returns as the upper bound the size of the requested dimension or the size of all dimensions in a rank-one array.

When the `ARRAY` argument is an array name or a structure component with no section subscript list, there are four cases to distinguish depending on the array specifier for the name. The following subsections describe these four cases.

Explicit-shape specifier
7.2.8.4.1

The LBOUND and UBOUND functions return the declared lower and upper bounds of the array name or the structure component with no section subscript list.

```
INTEGER  A(2:10, 11:12)

TYPE PASSENGER_INFO
   INTEGER NUMBER
   INTEGER TICKET_IDS(2:500)
END TYPE PASSENGER_INFO

TYPE(PASSENGER_INFO) PAL, MANY(3:10)
```

LBOUND(A) has the value (/ 2, 11 /), and UBOUND(A, 1) has the value 10. LBOUND(PAL%TICKET_IDS) has the value (/ 2 /) and UBOUND(MANY%TICKET_IDS(2), 1) has the value 10.

Assumed-shape specifier
7.2.8.4.2

The name is a dummy argument whose extents are determined by the corresponding actual argument. The dummy argument may have its lower bound in a particular dimension specified but if not, the lower bound is defined to be 1. The LBOUND function returns these lower bounds. The upper bound for a particular dimension is the extent of the actual argument in that dimension, if no lower bound is specified for the dummy argument. It is the extent minus 1 plus the lower bound if a lower bound is specified. The UBOUND function returns these upper bounds.

```
REAL  C(2:10, 11:12)
   . . .
CALL  S(C(4:8, 7:9))
CONTAINS
  SUBROUTINE S(A)
    REAL  A(:, 2:)
       . . .
    ! Reference to LBOUND(A)  and  UBOUND(A)
       . . .
```

Inside the body of subroutine S, LBOUND(A) has the value (/ 1, 2 /), because the array starts at subscript position 1 by default in the first dimension and starts at subscript position 2 by declaration in the second dimension. UBOUND(A) has the value (/ 5, 4 /), because there are five subscript positions (4 to 8) in the first dimension of the actual argument corresponding to A, and three subscript positions (7 to 9) in the second dimension of the same actual argument and the subscripts are specified to start at 2 by the declaration of the dummy argument A.

Assumed-size specifier

7.2.8.4.3

The name is a dummy argument whose upper and lower bounds in all but the last dimension are declared for the dummy argument. The lower bound for the last dimension may be specified in the assumed-size specifier but, if absent, the lower bound is 1. The LBOUND function returns these lower bounds. The upper bound for all dimensions except the last one is known to the subprogram but the upper bound in the last dimension is not known. The UBOUND function, therefore, must not be referenced with the first argument being the name of an assumed-size array and no second argument, or the first argument being the name of an assumed-size array and the second argument specifying the last dimension of the array. Otherwise, the UBOUND function returns the upper bounds as declared for all but the last dimension.

```
REAL  C(2:10, 11:12)
   . . .
CALL  S (C(4:8, 7:9))
CONTAINS
  SUBROUTINE  S (A)
    REAL  A(-2:2, *)
       . . .
 ! Reference to LBOUND(A, 1)  and  UBOUND(A(:, 2))
 ! A reference to UBOUND(A) would be illegal.
 ! A reference to UBOUND(A, 2) would be illegal.
    . . .
```

Inside the body of subroutine S, LBOUND(A, 1) has the value -2. UBOUND(A(:, 2)) has the value 5 because A(:,2) is an expression, which is an array section, not an array name, and has five elements in the first dimension.

Deferred-shape specifier
7.2.8.4.4

The name is the name of an allocatable array, an array pointer, or a structure component with one of its part references being a pointer array. As such, if the array or a part reference has not been allocated or associated with a target, the LBOUND and UBOUND functions must not be invoked with the ARRAY argument equal to such an array name. If it is an array pointer, either its target has been allocated by an ALLOCATE statement or its target has become associated with the pointer using a pointer assignment statement. In the former case, the LBOUND and UBOUND functions return the lower and upper bounds specified in the ALLOCATE statement. In the latter case, LBOUND is always 1 in pointer assignment, and UBOUND is the extent of the dimension. Note that for zero-sized dimensions, LBOUND is always 1 and UBOUND is always 0.

```
REAL, ALLOCATABLE ::  A(:, :)
    . . .
ALLOCATE  ( A(5, 7:9) )
    . . .
! Reference to LBOUND(A)  and  UBOUND(A)
    . . .
```

After the ALLOCATE statement above is executed, LBOUND(A) has the value (/ 1, 7 /), and UBOUND(A) has the value (/ 5, 9 /).

Special expressions
7.2.9

Expressions can appear in statements other than assignment statements, in particular in specification statements. In many cases, such expressions are restricted in some way; for example, the operands in expressions in a PARAMETER statement are essentially restricted to constants. The standard and this manual use specific terms for the various categories of expressions allowed in specific syntactic contexts. For example, the expressions that can be used in PARAMETER statements are called *initialization expressions* and can be evaluated at the time the program is compiled. Initialization expressions are restricted forms of constant expressions.

The expressions that can be used as array bounds and character lengths in specification statements are called *specification expressions*. These expressions are scalar and of type integer and can be evaluated on entry to the program unit at the time of execution. The remainder of this subsection describes and defines such limited expressions and summarizes where they can be used.

Constant expressions
7.2.9.1

A *constant expression* is one of the following constant values or is an expression consisting of intrinsic operators whose operands are any of the following constant values:

1. A literal or named constant, or a subobject of a constant where each subscript, section subscript, or starting and ending point of a substring range is a constant expression
2. An array constructor where every subexpression has primaries that are constant expressions or are implied-`DO` variables of the array constructor
3. A structure constructor where each component is a constant expression
4. An intrinsic function reference that can be evaluated at compile-time
5. A constant expression enclosed in parentheses

The restriction in item 4 to intrinsic functions that can be evaluated at compile-time eliminates the use of the intrinsic functions `PRESENT`, `ALLOCATED`, and `ASSOCIATED`. It also requires that each argument of the intrinsic function reference be a constant expression or a variable whose type parameters or bounds are known at compile-time. This restriction excludes, for example, named variables that are assumed-shape arrays, assumed-size arrays for inquiries requiring the size of the last dimension, and variables that are pointer arrays or allocatable arrays. For example, if an array `X` has explicit bounds in all dimensions, an inquiry such as `SIZE(X)` can be computed at compile-time, and `SIZE(X) + 10` is considered a constant expression.

Constant expressions can be used in any executable statement where general expressions (that is, unrestricted expressions) are permitted.

The following examples show constant expressions:

Expression	Meaning
`2`	An integer literal constant
`-7.5_LARGE`	A real literal constant where `LARGE` is a named integer constant
`(/ 7, (I, I = 1, 10) /)`	An array constructor
`RATIONAL(1, 2+J)`	A structure constructor where `RATIONAL` is a derived type and `J` is a named integer constant
`LBOUND(A,1)+3`	A reference to an inquiry intrinsic function where `A` is an explicit-shape array
`INT(N)`	An intrinsic function reference where `N` is a named constant
`KIND(X)`	An intrinsic function reference where `X` is a real variable with known type parameter
`REAL(10+I)`	An intrinsic function reference where `I` is a named integer constant
`COUNT(A)`	An intrinsic function where `A` is a named logical constant
`I/3.3 + J**3.3`	A numeric expression where `I` and `J` are named integer constants
`SUM(A)`	A reference to a transformational intrinsic function where `A` is a named integer array constant

7.2.9.2 Initialization expressions

An *initialization expression* is a constant expression restricted as follows:

- The exponentiation operator (`**`) is allowed only when the power (second operand) is of type integer; that is, `X ** Y` is allowed only if `Y` is of type integer .
- Subscripts, section subscripts, starting and ending points of substring ranges, components of structure constructors, and arguments of intrinsic functions must be initialization expressions.
- The elements of array constructors must be initialization expressions or implied- `DO`s for which the array constructor values and implied-`DO` parameters are expressions whose primaries are initialization expressions or implied-`DO` variables.
- An elemental intrinsic function in an initialization expression must have arguments of type integer or character, and must return a result of type integer or character.
- A transformational intrinsic function in an initialization expression must be one of the transformational intrinsic functions `REPEAT`, `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `TRANSFER`, and `TRIM`, and must have initialization expressions as arguments; this excludes the use of the transformational functions `ALL`, `ANY`, `COUNT`, `CSHIFT`, `DOT_PRODUCT`, `EOSHIFT`, `MATMUL`, `MAXLOC`, `MAXVAL`, `MINLOC`, `MINVAL`, `PACK`, `PRODUCT`, `SPREAD`, `SUM`, `TRANSPOSE`, and `UNPACK`.
- An inquiry intrinsic function is allowed, except that the arguments must either be initialization expressions or variables whose type parameters or bounds inquired about are not assumed, not defined by an `ALLOCATE` statement, or not defined by pointer assignment.
- Any subexpression enclosed in parentheses must be an initialization expression.

All but the last examples in subsection 7.2.9.1, page 280 are initialization expressions. The last are not because initialization expressions cannot contain functions that return results of type real (`REAL`, `LOG`), must not reference certain transformational functions (`COUNT`, `SUM`), or cannot use the exponentiation operator when the second operand is of type real.

The following are examples of initialization expressions:

Expression	Meaning
`SIZE(A, 1) * 4`	An integer expression where `A` is an array with an explicit shape
`KIND(0.0D0)`	An inquiry function with a constant argument
`SELECTED_REAL_KIND(6, 30)`	An inquiry function with constant arguments
`SELECTED_INT_KIND(2 * R)`	An inquiry function with an argument that is an initialization expression, where `R` is a previously declared named constant of type integer

Initialization expressions must be used in the following contexts:

- As initial values following the equal signs in `PARAMETER` statements and in type declaration statements with the `PARAMETER` attribute.
- As initial values following the equal signs in type declaration statements for variables.
- As expressions in structure constructors in `DATA` statement value lists.
- As kind type parameter values in type declaration statements; in this case, they also must be scalar and of type integer.
- As actual arguments for the `KIND` dummy argument of the conversion intrinsic functions `AINT`, `ANINT`, `CHAR`, `INT`, `LOGICAL`, `NINT`, `REAL`, `CMPLX`; in this case, they also must be scalar and of type integer.
- As case values in the `CASE` statement; in this situation, they must be scalar and of type integer, logical, or character.
- As subscript or substring range expressions of equivalence objects in an `EQUIVALENCE` statement; in this case, they must be scalar and of type integer .

Initialization expressions must be used for situations where the value of the expression is needed at compile time. Note that the initialization expressions do not include intrinsic functions that return values of type real, logical, or complex, or have arguments of type real, logical, or complex.

7.2.9.3 Specification expressions

A *specification expression* is a restricted expression that has a scalar value and is of type integer. Specification expressions are used as bounds for arrays and length parameter values for character entities in type declarations, attribute specifications, dimension declarations, and other specification statements (see Table 14, page 289).

Specification expressions are forms of restricted expressions (defined below), limited in type and rank. Briefly, a restricted expression is limited to constants and certain variables accessible to the scoping unit whose values can be determined on entry to the program unit before any executable statement is executed. For example, variables that are dummy arguments, are in a common block, are in a host program unit, or are in a module made accessible to the program unit can be evaluated on entry to a program unit. Array constructors, structure constructors, intrinsic function references, and parenthesized expressions made up of these primaries must depend only on restricted expressions as building blocks for operands in a restricted expression.

A *restricted expression* is an expression in which each operation is intrinsic and each primary is limited to one of the following:

- A constant or constant subobject.
- A variable that is a dummy argument.
- A variable that is in a common block.
- A variable made accessible from a module.
- A variable from the host program unit.
- An array constructor in which every expression has primaries that are restricted expressions or are implied-`DO` variables of the array constructor .

- A structure constructor in which each component is a restricted expression.
- An elemental intrinsic function whose result is of type integer or character and whose arguments are all restricted expressions of type integer or character.
- One of the transformational intrinsic functions (`REPEAT`, `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `TRANSFER`, or `TRIM`), in which each argument is a restricted expression of type integer or character (this excludes the use of the transformational functions `ALL`, `ANY`, `COUNT`, `CSHIFT`, `DOT_PRODUCT`, `EOSHIFT`, `MATMUL`, `MAXLOC`, `MAXVAL`, `MINLOC`, `MINVAL`, `PACK`, `PRODUCT`, `SPREAD`, `SUM`, `TRANSPOSE`, and `UNPACK`).
- An inquiry intrinsic function (except for `PRESENT`, `ALLOCATED`, and `ASSOCIATED`), in which each argument is one of the following:
  - A restricted expression
  - A variable whose bounds or type parameters inquired about are not assumed, not defined by an `ALLOCATE` statement, and not defined by a pointer assignment statement

  Any subscript, section subscript, and starting or ending point of a substring range is a restricted expression.
- A reference to an external function whose result is a nonpointer scalar intrinsic type.

  ANSI/ISO: The Fortran 90 standard does not specify restricted expressions in which a primary can be a reference to an external function whose result is a nonpointer scalar intrinsic type.

7.2.9.4 Initialization and specification expressions in declarations

The following rules and restrictions apply to the use of initialization and specification expressions in specification statements.

The type and type parameters of a variable or named constant in one of these expressions must be specified in a prior specification in the same scoping unit, in a host scoping unit, in a module scoping unit made accessible to the current scoping unit, or by the implicit typing rules in effect. If the variable or named constant is explicitly given these attributes in a subsequent type declaration statement, it must confirm the implicit type and type parameters.

If an element of an array is referenced in one of these expressions, the array bounds must be specified in a prior specification.

If a specification expression includes a variable that provides a value within the expression, the expression must appear within the specification part of a subprogram. For example, consider variable N in the following program segment:

```
INTEGER  N
COMMON   N
REAL  A (N)
```

N is providing a value that determines the size of the array A. This program segment must not appear in a main program but may appear in the specification part of a subprogram.

A prior specification in the above cases may be in the same specification statement, but to the left of the reference. For example, the following declarations are valid:

```
INTEGER, DIMENSION(4), PARAMETER :: A = (/ 4, 3, 2, 1 /)
REAL, DIMENSION(A (2)) :: B, C(SIZE(B))
```

B and C are of size 3 (the second element of the constant array A). The following declaration, however, is invalid because SIZE(E) precedes E:

```
REAL, DIMENSION(2) :: D(SIZE(E)), E
```

Uses of the various kinds of expressions 7.2.9.5

The various kinds of expressions may be somewhat confusing, and it can be difficult to remember where they can be used. To summarize the differences, subsection 7.2.4, page 243 specifies the most general kind of expression; the other kinds of expressions are restrictions of the most general kind. The classification of expressions forms two orderings, each from most general to least general, as follows:

- Expression, restricted expression, and specification expression
- Expression, constant expression, and initialization expression

The relationship between the various kinds of expression can be seen in the diagram in Figure 15.

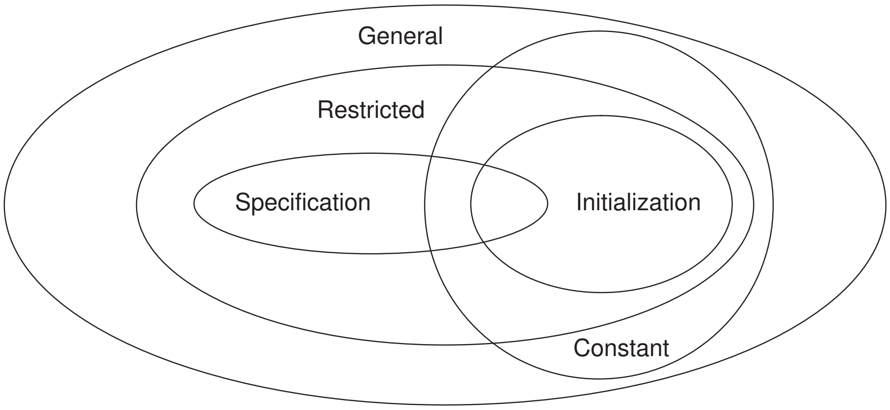

Figure 15. Relationships between the kinds of expressions

Initialization expressions are not a subset of specification expressions because the result of an initialization expression can be of any type, whereas the result of a specification expression must be of type integer and scalar . Also, specification expressions are not a subset of initialization expressions because specification expressions allow certain variables (such as dummy arguments and variables in common blocks) to be primaries, where as initialization expressions do not allow such variables.

Table 14 describes the differences between initialization and specification expressions. Table 15, page 291, summarizes where each kind of expression is used and gives the restrictions as to their type and rank when used in the various contexts. For example, Table 14 indicates that initialization and specification expressions are different in that initialization expressions can be array valued, whereas specification expressions are scalar. A consequence of this difference, as indicated in Table 15, is that an initialization expression is used in a type declaration statement or a `PARAMETER` statement to specify the value of a named constant array, whereas a specification expression is used to specify the bounds of an array in a declaration statement.

Section 7

Table 14. Differences and similarities between initialization and specification expressions

Property	Kind of expression: Initialization	Specification
Character result	Yes	No †
Integer result	Yes	Yes
Scalar result	Yes	Yes
Array result	Yes	No
Variables as primaries (limited to dummy arguments, common block objects, host objects, module objects)	No	Yes
Elemental intrinsic functions of type integer and character as primaries	Yes	Yes
Elemental intrinsic functions of type real, complex, logical, and derived type as primaries	No	No
Only constants as primaries	Yes	No
Only constant subscripts, strides, character lengths	Yes	No
One of the transformational intrinsic functions `REPEAT`, `RESHAPE`, `SELECTED_INT_KIND`, `SELECTED_REAL_KIND`, `TRANSFER`, or `TRIM` as primaries	Yes	Yes

(continued)

Table 14. Differences and similarities between initialization and specification expressions (continued)

Property	Kind of expression: Initialization	Specification
Inquiry intrinsic functions (not including `ALLOCATED`, `ASSOCIATED`, or `PRESENT`) as primaries	Yes	Yes
Reference to scalar external function with intrinsic type result††	No	Yes

† Expression results of type character are allowed if they are arguments of an intrinsic function.

†† This is a CF90 compiler extension to the Fortran 90 standard.

Table 15. Kinds of expressions and their uses

Context	Arb. expr.	Init. expr.	Spec. expr.	Type †	Rank ††
Bounds in declaration statement†††	No	No	Yes	I	Scalar
Lengths in declaration statement§	No	No	Yes	I	Scalar
Subscripts and substring ranges in `EQUIVALENCE` statement	No	Yes	No	I	Scalar
Values in `CASE` statement	No	Yes	No	I,L,C	Scalar
Kind parameters in declaration statement	No	Yes	No	I	Scalar
Kind arguments in intrinsics	No	Yes	No	I	Scalar
Initial value in `PARAMETER` and type declaration statement	No	Yes	No	Any	Any
`DATA` implied-`DO` parameters	No	§§	No	I	Scalar
Assignment	Yes	Yes	Yes	Any	Any
Subscripts in executable statement	Yes	Yes	Yes	I	≤1
Strides in executable statement	Yes	Yes	Yes	I	Scalar
Substring ranges in executable statement	Yes	Yes	Yes	I	Scalar
Expression in `SELECT CASE`	Yes	Yes	Yes	I,L,C	Scalar
`IF` statement	Yes	Yes	Yes	L	Scalar
Arithmetic `IF` statement	Yes	Yes	Yes	I,R	Scalar
`DO` statement	Yes	Yes	Yes	I,R	Scalar
Mask in `WHERE` statement	Yes	Yes	Yes	L	Array

(continued)

Table 15. Kinds of expressions and their uses

Context	Arb. expr.	Init. expr.	Spec. expr.	Type †	Rank ††
Mask in WHERE construct	Yes	Yes	Yes	L	Array
IF−THEN statement	Yes	Yes	Yes	L	Scalar
ELSE−IF statement	Yes	Yes	Yes	L	Scalar
Output item list	Yes	Yes	Yes	Any	Any
I/O specifier values except character FMT= specifier§§§	Yes	Yes	Yes	I,C	Scalar
I/O FMT= specifier value	Yes	Yes	Yes	C	Any
RETURN statement	Yes	Yes	Yes	I	Scalar
Computed GO TO statement	Yes	Yes	Yes	I	Scalar
Array constructor implied-DO parameters	Yes	Yes	Yes	I	Scalar
Actual arguments	Yes	Yes	Yes	Any	Any
I/O implied-DO parameters	Yes	Yes	Yes	I,R	Scalar
Expressions in statement function definitions	Yes	Yes	Yes	Any	Scalar

† "Any" in this column means any intrinsic or derived type.
†† "Any" in this column means that the result may be a scalar or an array of any rank.
††† The relevant declaration statements are type declaration, component definition, DIMENSION, TARGET, and COMMON statements.
§ The relevant declaration statements are type declaration, component definition, IMPLICIT, and FUNCTION statements.
§§ A DATA implied-DO parameter may be an expression involving intrinsic operations with constants and DATA implied-DO variables as operands.
§§§ If the I/O FMT= specifier is not of type character, it must be a default integer variable.

Example 1: The expressions I*3, 1+2*J, 5*J/3, 1, and 10, in the following, are all expressions allowed in subscripts and implied-DO parameter expressions in an implied-DO list in a DATA statement:

```
DATA ((A(I*3), I = 1+2*J, 5*J/3), J = 1, 10) /.../
```

Example 2: An expression such as RADIX(I) is not allowed as a data implied- DO parameter or subscript of a DATA statement object.

An expression such as N, where N is a variable in the scoping unit that contains the DATA statement, is not allowed because N is neither a named constant nor an implied-DO variable in a containing implied-DO list.

Such special expressions in DATA statements are restricted forms of initialization expressions in the sense that the primaries must not include references to any intrinsic function. On the other hand, they are extended forms of initialization expressions in the sense that they permit the use of implied- DO variables that have the scope of the implied-DO list.

Interpretation of expressions

7.3

The interpretation of an expression specifies the value of the expression when it is evaluated. As with the rules for forming an expression, the rules for interpreting an expression are described from the bottom up, from the interpretation of constants, variables, constructors, and functions to the interpretation of each subexpression to the interpretation of the entire expression.

When an expression is interpreted, the value of each constant and variable is determined. After these are determined, the operations for which the variables and constants are interpreted are interpreted in precedence order, and a value for the operation is determined by the interpretation rules for each operator. This repeats recursively until the entire expression is interpreted and a value is determined.

The interpretation rules for operations are of two sorts: rules for the intrinsic operations (intrinsic operators with operands of the intrinsic types specified by Table 12, page 264) and rules for the defined operations (provided by the programmer using function subprograms). Except for integer division, the intrinsic operations are interpreted by the usual mathematical method, subject to representation limitations imposed by a computer (for example, a finite range of integers, or finite precision of real numbers). The defined operations are interpreted by a function program that is specified in an interface block with a generic specifier of the form `OPERATOR (`*defined_operator* `)`.

The interpretation rules for an intrinsic or a defined operation are independent of the context in which the expression occurs. That is, the type, type parameters, and interpretation of any expression do not depend on any part of a larger expression in which it occurs.

Interpretation of the intrinsic operations

7.3.1

When the arguments of the intrinsic operators satisfy the requirements of Table 12, page 264, the operations are intrinsic and are interpreted in the usual mathematical way as described in Table 16, except for integer division. For example, the binary operator * is interpreted as the mathematical operation multiplication and the unary operator – is interpreted as negation.

Table 16. Interpretation of the intrinsic operations

Use of operator			Interpretation
x_1	`**`	x_2	Raise x_1 to the power x_2
x_1	`/`	x_2	Divide x_1 by x_2
x_1	`*`	x_2	Multiply x_1 by x_2
x_1	`–`	x_2	Subtract x_2 from x_1
	`–`	x_2	Negate x_2
x_1	`+`	x_2	Add x_1 and x_2
	`+`	x_2	Same as x_2
x_1	`//`	x_2	Concatenate x_1 with x_2
x_1	`.LT.`	x_2	x_1 less than x_2

Table 16. Interpretation of the intrinsic operations (continued)

Use of operator			Interpretation
x_1	`<`	x_2	x_1 less than x_2
x_1	`.LE.`	x_2	x_1 less than or equal to x_2
x_1	`<=`	x_2	x_1 less than or equal to x_2
x_1	`.GT.`	x_2	x_1 greater than x_2
x_1	`>`	x_2	x_1 greater than x_2
x_1	`.GE.`	x_2	x_1 greater than or equal to x_2
x_1	`>=`	x_2	x_1 greater than or equal to x_2
x_1	`.EQ.`	x_2	x_1 equal to x_2
x_1	`==`	x_2	x_1 equal to x_2
x_1	`.NE.`	x_2	x_1 not equal to x_2
x_1	`/=`	x_2	x_1 not equal to x_2
	`.NOT.`	x_2	True if x_2 is false
x_1	`.AND.`	x_2	True if x_1 and x_2 are both true
x_1	`.OR.`	x_2	True if x_1 and/or x_2 is true
x_1	`.NEQV.`	x_2	True if either x_1 or x_2 is true, but not both
x_1	`.XOR.`	x_2	1 if corresponding bits differ; 0 otherwise (bitwise exclusive `.OR.`)
x_1	`.EQV.`	x_2	True if both x_1 or x_2 are true or both are false

7.3.1.1 Interpretation of numeric intrinsic operations

Except for exponentiation to an integer power, when an operand for a numeric intrinsic operation does not have the same type or type parameters as the result of the operation, the operand is converted to the type, type parameter , and shape of the result and the operation is then performed. For exponentiation to an integer power, the operation can be performed without the conversion of the integer power, say, by developing binary powers of the first operand and multiplying them together to obtain an efficient computation of the result.

For integer division, when both operands are of type integer, the result is of type integer, but the mathematical quotient is often not an integer. In this case, the result is specified to be the integer value closest to the quotient and between zero and the quotient inclusively.

For exponentiation, there are three cases that need to be further described. When both operands are of type integer, the result is of type integer; when x_2 is negative, the operation x_1`**`x_2 is interpreted as the quotient `1/(`x_1`**ABS(`x_2`))`. Note that it is subject to the rules for integer division. For example, `4**(-2)` is `0`.

The second case occurs when the first operand is a negative value of type integer or real and the second operand is of type real. A program is invalid if it causes a reference to the exponentiation operator with such operands. For example, a program that contains the expression `(-1.0) ** 0.5` is an invalid program.

The third case occurs when the second operand is of type real or of type complex. In this case, the result returned is the principal value of the mathematical power function $x_1^{x_2}$.

7.3.1.2 Interpretation of standard nonnumeric intrinsic operations

There is only one intrinsic character operation: concatenation. For this operation, the operands must be of type character. The length parameter values can be different. The result is of type character with a length type parameter value equal to the sum of the lengths of the operands. The result consists of the characters of the first operand in order followed by those of the second operand in order. For example, `'Fortran ' // ' 90'` yields the result `'Fortran 90'`.

The intrinsic relational operations perform comparison operations for character and most numeric operands. For these operations, the operands must both be of numeric type or both be of character type. The kind type parameter values of the operands of the numeric types can be different and the lengths of character operands can be different. Complex operands must only be compared for equality and inequality; the reason is that complex numbers are not totally ordered. The result in all cases is of type default logical.

When the operands of an intrinsic relational operation are both numeric, but of different types or type parameters, each operand is converted to the type and type parameters they would have if the two operands were being added. Then, the operands are compared according to the usual mathematical interpretation of the particular relational operator.

When the operands are both of type character, the shorter one is padded on the right with blank padding characters until the operands are of equal length. Then, the operands are compared one character at a time in order, starting from the leftmost character of each operand until the corresponding characters differ. The first operand is less than or greater than the second operand according to whether the characters in the first position where they differ are less than or greater than each other. The operands are equal if both are of zero length or all corresponding characters are equal, including the padding characters.

There is no ordering defined for logical values. However, logical values can be compared for equality and inequality by using the logical equivalence and not equivalence operators `.EQV.` and `.NEQV.` That is, `L1 .EQV. L2` is true when `L1` and `L2` are both true or both false and is false otherwise; `L1 .NEQV. L2` is true if either `L1` or `L2` is true (but not both true) and is false otherwise.

The intrinsic logical operations perform many of the common operations for logical computation. For these operations, the operands must both be of logical type but can have different kind type parameters. If the kind type parameter values are the same, the kind type parameter value of the result is the kind type parameter value of the operands. If the kind type parameter values are different, the kind type parameter value is the larger of the two kind type parameter values. The values of the result in all cases are specified in Table 17.

Table 17. The values of operations involving logical operators

x_1	x_2	.NOT.x_1	x_1.AND.x_2	x_1.OR.x_2	x_1.EQV.x_2	x_1.NEQV.x_2
True	True	False	True	True	True	False
True	False	False	False	True	False	True
False	True	True	False	True	False	True
False	False	True	False	False	True	False

Interpretation of intrinsic operations with array operands
7.3.1.3

Each of the intrinsic operations can have array operands; however, for the binary intrinsic operations, the operands must both be of the same shape, if both are arrays. When one operand is an array and the other is a scalar, the operation behaves as if the scalar operand were broadcast to an array of the result shape and the operation performed.

For both the unary and binary intrinsic operators, the operation is interpreted element-by-element; that is, the scalar operation is performed on each element of the operand or operands. For example, if `A` and `B` are arrays of the same shape, the expression `A * B` is interpreted by taking each element of `A` and the corresponding element of `B` and multiplying them together using the scalar intrinsic operation `*` to determine the corresponding element of the result. Note that this is not the same as matrix multiplication. As a second example, the expression `-A` is interpreted by taking each element of `A` and negating it to determine the corresponding element of the result.

For intrinsic operations that appear in masked assignment statements (in `WHERE` blocks, `ELSEWHERE` blocks, or in a `WHERE` statement), the scalar operation is performed only for those elements selected by the logical mask expression.

Note that there is no order specified for the interpretation of the scalar operations. A processor is allowed to perform them in any order, including all at the same time.

7.3.1.4 Interpretation of intrinsic operations with pointer operands

The intrinsic operations can have operands with the POINTER attribute. In such cases, each pointer must be associated with a target that is defined, and the value of the target is used as the operand. The target can be scalar or array-valued; the rules for interpretation of the operation are those appropriate for the operand being a scalar or an array, respectively.

Recall that an operand can be a structure component that is the component of a structure variable that is itself a pointer. In this case, the value used for the operand is the named component of the target structure associated with the structure variable. For example, consider the following declarations and assume that the pointer PTR is associated with T:

```
TYPE RATIONAL
   INTEGER :: N, D
END TYPE

TYPE(RATIONAL), POINTER :: PTR
TYPE(RATIONAL), TARGET  :: T
```

If PTR%N appears as an operand, its value is the component N of the target T, namely T%N.

7.3.2 Interpretation of defined operations

The interpretation of a defined operation is provided by a function subprogram with an OPERATOR interface (see the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903). When there is more than one function with the same OPERATOR interface, the function giving the interpretation of the operation is the one whose dummy arguments match the operands in order, types, kind type parameters, and rank. For example, for the operation A .PLUS. B, where A and B are structures of the derived type RATIONAL, the following interface specifies that the function RATIONAL_PLUS provides the interpretation of this operation.

Example:

```
INTERFACE OPERATOR(.PLUS.)

   FUNCTION RATIONAL_PLUS(L, R)
      USE RATIONAL_MODULE
      TYPE(RATIONAL), INTENT(IN) :: L, R
      TYPE(RATIONAL)            :: RATIONAL_PLUS
   END FUNCTION RATIONAL_PLUS

   FUNCTION LOGICAL_PLUS(L, R)
      LOGICAL, INTENT(IN) :: L, R
      LOGICAL              :: LOGICAL_PLUS
   END FUNCTION LOGICAL_PLUS

END INTERFACE
```

A defined operation is declared by using a function with one or two dummy arguments. (Note that the function can be an entry in an external or module function.)

The dummy arguments to the function represent the operands of the operation. If there is only one, the operation is a unary operation; otherwise it is a binary operation. For a binary operation, the first argument is the left operand and the second is the right operand.

There must be an interface block for the function with the generic specifier of the form `OPERATOR (`*defined_operator*`)`.

The types and kind type parameters of the operands in the expression must be the same as those of the dummy arguments of the function.

The rank of the operands in the expression must match the ranks of the corresponding dummy arguments of the function.

One of the following conditions must be true:

- One of the dummy arguments must be of a derived type
- Both of the dummy arguments are of intrinsic type but do not match the types and kind type parameters for intrinsic operations as specified in Table 12, page 264.

The requirement that the shapes of the dummy arguments and operands match implies that the defined operators are never elemental; that is, if a defined operation is specified for a scalar operator, it does not apply to array operands of the same shape as is the case for intrinsic operations. Note also that the operands of a defined operation need not be in shape conformance as is required for the intrinsic operations.

As with the intrinsic operations, the type, type parameters, and interpretation of a defined operation are independent of the context of the larger expression in which the defined operation appears. The interpretation of the same defined operation in different contexts is the same, but the results can be different because the results of the procedure being invoked may depend on values that are not operands and that are different for each invocation.

The relational operators `==`, `/=`, `>`, `>=`, `<`, and `<` are synonyms for the operators `.EQ.`, `.NE.`, `.GT.`, `.GE.`, `.LT.`, and `.LE.`, even when they are defined operators. It is invalid, therefore, to have an interface block for both `==` and `.EQ.`, for example, for which the order, types, type parameters, and rank of the dummy arguments of two functions are the same.

Defined operations are either unary or binary. An existing unary operator (that is, one that has the same name as an intrinsic operator) cannot be defined as a binary operator unless it is also a binary operator. Similarly, an existing binary operator cannot be defined as a unary operator unless it is also a unary operator. However, a defined operator, `.PLUS.` say, (that is, one that does not have a name that is the same as an intrinsic operator) can be defined as both a unary and binary operator.

Evaluation of expressions

7.4

The form of the expression and the meaning of the operations establish the interpretation; once established, a compiler can evaluate the expression in any way that provides the same interpretation with one exception; parentheses specify an order of evaluation that cannot be modified. This applies to both intrinsic operations and defined operations.

Essentially , two sorts of alternative evaluations are allowed:

- The rearrangement of an expression that yields an equivalent expression; for example, `A + B + C` can be evaluated equivalently as `A + (B + C)` and would improve the efficiency of the compiled program if `B + C` were a subexpression whose value had already been computed.

- The partial evaluation of an expression because the value of the unevaluated part can be proven not to affect the value of the entire expression. For example, when one operand of a disjunction (`.OR.` operator) is known to be true, the other operand need not be evaluated to determine the result of the operation. To be specific, the operand `A * B < C` need not be evaluated in the expression `A < B .OR. A * B < C` if `A < B` is true. This freedom for a compiler to use alternative equivalent evaluations permits the compiler to produce code that is more optimal in some sense (for example, fewer operations, array operations rather than scalar operations, or a reduction in the use of registers or work space), and thereby produce more efficient executable code.

Possible alternative evaluations

7.4.1

Before describing in more detail the possible evaluation orders, four basic issues need to be addressed, namely, definition of operands, well-defined operations, functions (and defined operations) with side effects, and equivalent interpretations.

Definition status is described in detail in the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903. For the purpose of evaluation of expressions, it is required that each operand is defined, including all of its parts, if the operand is an aggregate (an array, a structure, or a string). If the operand is a subobject (part of an array, structure, or string), only the selected part is required to be defined. If the operand is a pointer, it must be associated with a target that is defined. An integer operand must be defined with an integer value rather than a statement label.

For the numeric intrinsic operations, the operands must have values for which the operation is well-defined. For example, the divisor for the division operation must be nonzero, and the result of any of the numeric operations must be within the exponent range for the result data type;

otherwise, the program is not standard conforming. Other cases include limitations on the operands of the exponentiation operation `**`: for example, a zero-valued first operand must not be raised to a nonpositive second operand; and a negative-valued first operand of type real cannot be raised to a real power.

The third issue is functions with side effects. In Fortran, functions are allowed to have side effects; that is, they are allowed to modify the state of the program so that the state is different after the function is invoked than before it is invoked. This possibility potentially affects the equivalence of two schemes for evaluating an expression, particularly if the function modifies objects appearing in other parts of the expression. However, Fortran prohibits the formation of statements with these kinds of side effects. That is, a function (or defined operation) within a statement is not permitted to change any entity in the same statement. Exceptions are those statements that have statements within them, for example, an `IF` statement or a `WHERE` statement. In these cases, the evaluation of functions in the logical expressions in parentheses after the `IF` keyword or `WHERE` keyword are allowed to affect objects in the statement following the closing right parenthesis. For example, if `F` and `G` are functions that change their actual argument `I`, the following statements are valid, even though `I` is changed when the functions are evaluated:

```
IF (F(I))  A = I
WHERE (G(I))  B = I
```

The following statements are examples of statements that are not valid because `F` and `G` change `I`, which is used elsewhere in the same statement:

```
A(I) = F(I)
Y = G(I) + I
```

It is also not valid for there to be two function references in a statement, if each causes a side effect and the order in which the functions are invoked yields a different final status, even though nothing in the statement is changed.

The fourth issue is equivalent interpretation. For the numeric intrinsic operations, the definition of equivalent interpretation is defined as being mathematical equivalence of the expression, not computational equivalence. Mathematical equivalence assumes exact arithmetic (no rounding errors and infinite exponent range) and thus assumes the rules of commutativity, associativity , and distributivity as well as other rules that can be used to determine equivalence (except that the order of operations specified by parentheses must be honored). Under these assumptions, two evaluations are mathematically equivalent if they yield the same values for all possible values of the operands. `A + B + C` and `A + (B + C)` are thus mathematically equivalent but are not necessarily computationally equivalent because of possible different rounding errors. On the other hand, `I/2` and `0.5 * I` (where `I` is an integer) is a mathematical difference because of the special Fortran definition of integer division.

Table 18 gives examples of equivalent evaluations of expressions where `A`, `B`, and `C` are operands of type real or complex, and `X`, `Y`, and `Z` are of any numeric type. All of the variables are assumed to be defined and have values that make all of the operations in this table well-defined.

Table 18. Equivalent evaluations for numeric intrinsic operations

Expression	Equivalent evaluations
`X+Y`	`Y+X`
`X*Y`	`Y*X`
`-X+Y`	`Y-X`
`X+Y+Z`	`X+(Y+Z)`
`X-Y+Z`	`X-(Y-Z)`
`X*A/Z`	`X*(A/Z)`
`X*Y-X*Z`	`X*(Y-Z)`
`A/B/C`	`A/(B*C)`
`A/5.0`	`0.2*A`

Table 19 provides examples of alternative evaluations that are not valid and are not mathematically equivalent to the original expression. In addition to the operands of the same names used in Table 18, Table 19 uses `I` and `J` as operands of type integer. Recall that when both operands of the division operator are of type integer, a Fortran integer division truncates the result toward zero to obtain the nearest integer quotient.

Table 19. Nonequivalent evaluations of numeric expressions

Expression	Prohibited evaluations
`I/2`	`0.5*I`
`X*I/J`	`X*(I/J)`
`I/J/A`	`I/(J*A)`
`(X+Y)+Z`	`X+(Y+Z)`
`(X*Y)-(X*Z)`	`X*(Y-Z)`
`X*(Y-Z)`	`X*Y-X*Z`

Partial evaluations
7.4.2

For character, relational, and logical intrinsic operations, the definition of the equivalence of two evaluations is that, given the same values for their operands, each evaluation produces the same result. The definition for equivalence of two evaluations of the same defined operation also requires the results to be the same; note that this definition is more restrictive than for the numeric intrinsic operations, because only mathematical equivalence need be preserved for numeric operations. As described for numeric intrinsic operations, a compiler can choose any evaluation scheme equivalent to that provided by the interpretation. Table 20 gives some equivalent schemes for evaluating a few example expressions. For these examples, `I` and `J` are of type integer; `L1`, `L2`, and `L3` are of type logical; and `C1`, `C2`, and `C3` are of type character of the same length. All of the variables are assumed to be defined.

Table 20. Equivalent evaluations of other expressions

Expression	Equivalent evaluations
`I .GT. J`	`(I-J) .GT. 0`
`L1 .OR. L2 .OR. L3`	`L1 .OR. (L2 .OR. L3)`
`L1 .AND. L1`	`L1`
`C3 = C1//C2`	`C3=C1` (`C1`, `C2`, `C3` all of the same length)

These rules for equivalent evaluation schemes allow the compiler to not evaluate any part of an expression that has no effect on the resulting value of the expression. Consider the expression `X*F(Y)`, where `F` is a function and `X` has the value 0. The result will be the same regardless of the value of `F(Y)`; therefore, `F(Y)` need not be evaluated. This shortened evaluation is allowed in all cases, even if `F(Y)` has side effects. In this case every data object that `F` could affect is considered to be undefined after the expression is evaluated (that is, it does not have a predictable value).

The appearance of an array element, an array section, or a character substring reference requires, in most cases, the evaluation of the expressions that are the subscripts, strides, or substring ranges. The type or type parameters of the containing expression are not affected by the evaluation of such subscript, stride, or substring range expressions. It is not necessary for these expressions to be evaluated, if the array section can be shown to be zero-sized or the substring can be shown to be of a zero-length by other means. For example, in the expression `A(1:0) +` `B(`$expr_1$:$expr_2$`)`, $expr_1$ and $expr_2$ need not be evaluated because the conformance rules for intrinsic operations require that the section of `B` be zero-sized.

The type and type parameters, if any, of a constructor are not affected by the evaluation of any expressions within the constructor.

Parentheses within the expression must be honored. This is particularly important for computations involving numeric values in which rounding errors or range errors may occur or for computations involving functions with side effects.

Assignment

7.5

The most common use of the result of an expression is to give a value to a variable. This is done with an assignment statement. For example:

```
RUG = BROWN + 2.34 / TINT
```

The forms of intrinsic assignment, defined assignment, and masked array assignment are the same, and they are defined as follows:

R735	*assignment_stmt*	**is**	*variable* = *expr*

Assignment establishes a value for the variable on the left of the assignment symbol in an assignment statement. Execution of the assignment statement causes the expression to be evaluated (by performing the computation indicated), and then the value of the expression is assigned to the variable. If the variable has subscripts, section subscripts, or a substring range, the execution of the assignment statement must behave as if they were evaluated before any part of the value is assigned.

There are four forms of the assignment statement: intrinsic assignment, defined assignment, pointer assignment, and masked array assignment.

An assignment statement is a defined assignment if the following conditions are true:

- There is a subroutine subprogram with an assignment interface of the form `ASSIGNMENT (=)`.
- The types, kind type parameters, and ranks (if arrays) of the variable and expression match in order the dummy arguments of the subroutine with the assignment interface.

An assignment statement is a masked array assignment if it appears in a `WHERE` construct or `WHERE` statement; otherwise, it is an intrinsic or defined assignment.

The form of the pointer assignment statement is similar to the assignment statement except that the assignment operator is => instead of =.

The rules and restrictions for each of these forms of assignment are different and are described in the subsections below for each form of assignment.

The following examples show the four forms of assignment:

Expression	Meaning
`X = X + 1.0`	Intrinsic assignment for reals
`CHAR(1:4) = "A123"`	Intrinsic assignment for characters
`STUDENT = B_JONES`	Intrinsic assignment for structures if `STUDENT` and `B_JONES` are of the same derived type
`STRING = "Brown"`	Defined assignment for structure if `STRING` is of derived type and an interface exists that defines the operator = for the types `STRING` and character
`WHERE (Z /= 0.0)` `  A = B / Z` `END WHERE`	Masked array assignment
`PTR => X`	Pointer assignment

Intrinsic assignment
7.5.1

Intrinsic assignment can be used to assign a value to a nonpointer variable of any type or to the target associated with a pointer variable. The assignment statement defines or redefines the value of the variable or the target, as appropriate. The value is determined by the evaluation of the expression on the right side of the equal sign.

The types and kind parameters of the variable and expression in an intrinsic assignment statement must be of the types given in Table 21.

Table 21. Types of the variable and expression in an intrinsic assignment

Type of the variable	Type of the expression
Integer	Integer , real, complex, Boolean, Cray pointer
Real	Integer , real, complex, Boolean
Complex	Integer , real, complex
Character	Character
Logical	Logical
Cray pointer	Cray pointer, integer, Boolean
Derived type	Same derived type as the variable

Note that Boolean and Cray pointer types are Cray Research extensions to the Fortran 90 standard.

If the variable is an array, the expression must either be a scalar or an array of the same shape as the variable. If the variable is a scalar, the expression must be a scalar. The shape of the variable can be specified in specification statements if it is an explicit-shape array. The shape of the variable can be determined by the section subscripts in the variable, by an actual argument if it is a assumed-shape array, or by an `ALLOCATE` statement or a pointer assignment statement if it is a deferred-shape array. It must not be an assumed-size array unless there is a vector subscript, a scalar subscript, or a section subscript containing an upper bound in the last dimension of the array. The shape of the expression is determined by the shape of the operands, the operators in the expression, and the functions referenced in the expression. A complete description of the shape of an expression appears in subsection 7.2.8.3, page 275.

If the variable is a pointer, it must be associated with a target; the assignment statement assigns the value of the expression to the target of the pointer. The pointer can be associated with a target that is an array; the pointer determines the rank of the array, but the extents in each dimension are that of the target.

The evaluation of the expression on the right-hand side of the equal sign, including subscript and section subscript expressions that are part of the expression and part of the variable, must be performed before any portion of the assignment is performed. Before the assignment begins, any necessary type conversions are completed if the variable has a different numeric type or type parameter from the expression. The conversion is the same as that performed by the conversion intrinsic functions `INT`, `REAL`, `CMPLX`, and `LOGICAL`, as specified in Table 22. Note that if the expression is of type Boolean or is a BOZ constant, no conversion is done.

ANSI/ISO: The Fortran 90 standard does not specify type Boolean or BOZ constants in assignment statements.

Table 22. Conversion performed on an expression before assignment

Type of the variable	Value assigned
Integer	`INT` (*expr*, `KIND` (*variable*))
Real	`REAL` (*expr*, `KIND` (*variable*))
Complex	`CMPLX` (*expr*, `KIND` (*variable*))
Logical	`LOGICAL` (*expr*, `KIND` (*variable*))

An expression can use parts of the variable that appear on the left side of an assignment statement. For example, in evaluating a character string expression on the right-hand side of an assignment, the values in the variable on the left-hand side can be used, as in the following example:

```
DATE(2:5) = DATE(1:4)
```

If the variable and expression are of character type with different lengths, the assignment occurs as follows:

- If the length of the variable is less than that of the expression, the value of the expression is truncated from the right.
- If the length of the variable is greater than the expression, the value of the expression is filled with blanks on the right.

The evaluation of expressions in the variable on the left-hand side, such as subscript expressions, has no effect on, nor is affected by, the evaluation of the expression on the right-hand side, which is evaluated completely first. As usual, this requirement that the expression on the right be evaluated first is specifying the semantics of the statement and does not imply that an implementation must perform the computation in this way if there is an equivalent order that computes the same result.

When a scalar is assigned to an array, the assignment behaves as if the scalar is broadcast to an array of the shape of the variable; it is then in shape conformance with the variable. In the following example, all ten elements of the array `A` are assigned the value `1.0`:

```
REAL A(10)
A = 1.0
```

Array assignment is element-by-element, but the order is not specified. Assume that `A` and `B` are real arrays of size `10`, and the whole array assignment is as follows:

```
A = B
```

The first element of `B` would be assigned to the first element of `A`, the second element of `B` would be assigned to the second element of `A`, and this would continue element-by-element for `10` elements. The assignment of elements, however , may be performed in any order.

For derived-type intrinsic assignment, the derived types of the variable and the expression must be the same. Derived-type intrinsic assignment is performed component-by-component following the above rules, except when a component is a pointer. For pointer components, pointer assignment between corresponding components is used.

7.5.2 Defined assignment

A *defined assignment* is an assignment operation provided by a subroutine with an assignment interface `ASSIGNMENT (=)`. When the variable and expression in the assignment statement are of intrinsic types and do not satisfy the type matching rules described in Table 21, page 309, or are of derived type, a defined assignment operation will be used, provided the assignment interface and subroutine are accessible. For example, a defined assignment may apply when an integer object is to be assigned to a logical variable, provided a subroutine with a generic assignment interface is accessible. Assignment thus can be extended to types other than the intrinsic types or can replace the intrinsic assignment operation for derived types, if the programmer defines the rules for this assignment in a subroutine. For more information on the assignment interface, see the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

An assignment operation is declared by using a subroutine with two dummy arguments.

The dummy arguments to the subroutine represent the variable and the expression, in that order.

There must be an interface block for the subroutine with the generic specifier of the form `ASSIGNMENT(=)`.

The types and kind type parameters of the variable and expression in the assignment statement must be the same as those of the dummy arguments.

The rank of the variable and the expression in the assignment must match the ranks of the corresponding dummy arguments.

One of the following conditions must be true:

- One of the dummy arguments must be of a derived type
- Both of the dummy arguments are of intrinsic type but do not match the types and kind type parameters for intrinsic operations as specified in Table 21, page 309.

Example:

```
INTERFACE  ASSIGNMENT (=)

   SUBROUTINE  RATIONAL_TO_REAL(L, R)
      USE  RATIONAL_MODULE
      TYPE(RATIONAL), INTENT(IN)    :: R
      REAL, INTENT(OUT)             :: L
   END SUBROUTINE  RATIONAL_TO_REAL
   SUBROUTINE  REAL_TO_RATIONAL(L, R)
      USE  RATIONAL_MODULE
      REAL, INTENT(IN)               :: R
      TYPE(RATIONAL), INTENT(OUT)    :: L
   END SUBROUTINE  REAL_TO_RATIONAL

END INTERFACE
```

The preceding interface block specifies two defined assignments for two assignment operations in terms of two external subroutines, one for assignment of objects of type RATIONAL to objects of type real, and the other for assignment of objects of type real to objects of type RATIONAL. With this interface block, the following assignment statements are defined:

```
REAL  R_VALUE
TYPE(RATIONAL)  RAT_VALUE

R_VALUE = RATIONAL(1, 2)
RAT_VALUE = 3.7
```

The effect of the defined assignment on variables in the program is determined by the referenced subroutine.

Pointer assignment
7.5.3

A pointer is a variable with the POINTER attribute that points to another object. The term *pointer association* is used for the concept of "pointing to" and the term *target* is used for the object associated with a pointer.

A pointer assignment associates a pointer with a target. If the target is disassociated or undefined, the pointer becomes disassociated or undefined according to the status of the target.

Once a pointer assignment has been executed, the association status of the pointer remains unchanged until one of the following events occurs:

- Another pointer assignment statement is executed that redefines the pointer .
- An `ALLOCATE`, `DEALLOCATE`, or `NULLIFY` statement is executed that redefines the pointer.
- A `RETURN` statement is executed. This changes the association status only if the pointer is local to the subprogram containing the return and the pointer does not have the `SAVE` attribute.

The pointer assignment statement is defined as follows:

R736	*pointer_assignment_stmt*	**is**	*pointer_object* => *target*
R630	*pointer_object*	**is**	*variable_name*
		or	*structure_component*
R737	*target*	**is**	*variable*
		or	*expr*

If the pointer object is a variable name, the name must have the `POINTER` attribute. If the pointer object is a structure component, the component must have the `POINTER` attribute.

The form of the expression permitted as a target is severely limited.

If the target is a variable, then it must have one of the following characteristics:

- It must be the `TARGET` attribute.
- It must be the component of a structure, the element of an array variable, or the substring of a character variable that has the `TARGET` attribute.
- It must have the `POINTER` attribute.

The type, type parameters (including length, if character), and rank of the target must be the same as the pointer object.

If the variable on the right of => has the TARGET attribute, the pointer object on the left of => becomes associated with this target.

If the variable on the right of => has the POINTER attribute and is associated, the pointer object on the left of => points to the same data that the target points to after the pointer assignment statement is executed.

If the variable on the right of => has the POINTER attribute and is disassociated, the pointer object on the left of => becomes disassociated.

If the variable on the right of => has the POINTER attribute and has an undefined association status, the association status of the pointer object on the left of => becomes undefined.

A pointer assignment statement terminates any previous association for that pointer and creates a new association.

If the pointer object is a deferred-shape array, the pointer assignment statement establishes the extents for each dimension of the array, unless the target is a disassociated or undefined pointer. Except for the case of a disassociated or undefined pointer, the extents are those of the target. For example, if the following statements have been processed, the extents of P1 are those of T, namely 11 and 20, but those of P2 are 1 and 10, because T(:) has a section subscript list:

```
INTEGER, TARGET :: T(11:20)
INTEGER, POINTER :: P1(:), P2(:)
P1 => T
P2 => T(:)
```

The target must not be a variable that is an assumed-size array. If it is an array section of an assumed-size array, the upper bound for the last dimension must be specified.

If the target is an array section, it must not have a vector subscript.

If the target is an expression, it must deliver a pointer result. This implies that the expression must be a user-defined function reference or defined operation that returns a pointer (there are no intrinsic operations or functions that return results with the POINTER attribute). This also implies that a pointer can never point at a constant because constants cannot have the TARGET attribute.

If the target of a pointer cannot be referenced or defined, the pointer must not be referenced or defined.

If a structure has a component with the `POINTER` attribute and the structure is assigned a value using an intrinsic derived-type assignment, pointer assignment is used for each component with the `POINTER` attribute. Also, defined assignment may cause pointer assignment between some components of a structure.

Note that when a pointer appears on the right side of => in a pointer assignment, the pointer on the left side of => is defined or redefined to be associated with the target on the right side of the =>; neither the pointer on the right nor its target are changed in any way.

General examples:

```
MONTH => DAYS(1:30)
PTR => X(:, 5)
NUMBER => JONES % SOCSEC
```

Example 1: In this example, the target is another pointer:

```
REAL, POINTER :: PTR, P
REAL, TARGET :: A
REAL  B
A = 1.0
P => A
PTR => P
B = PTR + 2.0
```

The previous program segment defines `A` with the value `1.0`, associates `P` with `A`; then `PTR` is associated with `A` as well (through `P`). The value assigned to `B` in the regular assignment statement is `3.0`, because the reference to `PTR` in the expression yields the value of the target `A` which is the value `1.0`.

Example 2: In this example, the target is an expression:

```
INTERFACE
   FUNCTION POINTER_FCN(X)
      REAL  X
      REAL, POINTER :: POINTER_FCN
   END FUNCTION
END INTERFACE

REAL, POINTER :: P
REAL  A

P => POINTER_FCN(A)
```

In this example, the function `POINTER_FCN` takes a real argument and returns a pointer to a real target. After execution of the pointer assignment statement, the pointer `P` points to this real target.

Pointers can become associated by using the `ALLOCATE` statement instead of a pointer assignment statement. Pointers can become disassociated by using the `DEALLOCATE` or `NULLIFY` statements, as well as with the pointer assignment statement.

A pointer can be used in an expression (see subsection 7.3.1.4, page 299, for the details). Briefly, any reference to a pointer in an expression, other than in a pointer assignment statement, or in certain procedure references, yields the value of the target associated with the pointer. When a pointer appears as an actual argument corresponding to a dummy argument that has the `POINTER` attribute, the reference is to the pointer and not the value. Note that a procedure must have an explicit interface if it has a dummy argument with the `POINTER` attribute. For information on explicit interfaces, see the *CF90 Fortran Language Reference Manual*, *Volume 2*, publication SR–3903.

Section 7

Masked array assignment
7.5.4

Sometimes it is desirable to assign only certain elements of one array to another array. To invert the elements of an array element-by-element, for example, you have to avoid elements that are 0. The masked array assignment is ideal for such selective assignment, as the following example using a WHERE construct illustrates:

```
REAL  A(10,10)
  ...
WHERE (A /= 0.0)
    RECIP_A = 1.0 / A    ! Assign only where the
                         !    elements are nonzero
ELSEWHERE
    RECIP_A = 1.0        ! Use the value 1.0 for
                         !    the zero elements
END WHERE
```

The first array assignment statement is executed for only those elements where the mask `A /= 0.0` is true. Next, the second assignment statement (after the ELSEWHERE statement) is executed for only those elements where the same mask is false. If the values of RECIP_A where A is 0 are never used, this example can be simply written by using the WHERE statement, rather than the WHERE construct, as follows:

```
WHERE (A /= 0.0) RECIP_A = 1.0 / A
```

A *masked array assignment* is an intrinsic assignment statement in a WHERE block, an ELSEWHERE block, or a WHERE statement for which the variable being assigned is an array. The WHERE statement and WHERE construct appear to have the characteristics of a control statement or construct such as the IF statement and IF construct. But there is a major difference: every assignment statement in a WHERE construct is executed, whereas at most one block in the IF construct is executed. Similarly, the assignment statement following a WHERE statement is always executed. For this reason, WHERE statements and constructs are discussed here under assignment rather than under control constructs.

In a masked array assignment, the assignment is made to certain elements of an array based on the value of a logical array expression serving as a mask for picking out the array elements. The logical array expression acts as an array-valued condition on the elemental intrinsic operations, functions, and assignment for each array assignment statement in the `WHERE` statement or `WHERE` construct.

As in an intrinsic array assignment, a pointer to an array can be used as the variable, and a pointer to a scalar or an array can be used as a primary in the expression. If the target of the pointer is an array, the target array is masked in the same manner as a nonpointer array used in a masked array assignment.

`WHERE` statement 7.5.4.1

The `WHERE` statement is defined as follows:

R738	*where_stmt*	**is**	`WHERE (` *mask_expr* `)` *assignment_stmt*

Evaluation of the logical mask expression results in a logical array, which is treated as a mask. The mask array must conform with the variable being defined in the assignment statement. Each element of the array on the left side of the array assignment statement is assigned a value from the expression on the right, if the corresponding element in the mask is true. Where the mask is false, the array elements are not assigned a value. Any elemental intrinsic operations or functions within the expression are evaluated only for the selected elements.

The expression in the array assignment statement can contain nonelemental function references. *Nonelemental function references* are references to any function or operation defined by a subprogram, or any intrinsic function that is a transformational or an inquiry function. All elements of the arguments of such functions and returned results (if arrays) are evaluated in full. If the result of the nonelemental function is an array and is an operand of an elemental operation or function, then only the selected elements are used in evaluating the remainder of the expression.

Example:

```
WHERE (TEMPERATURES > 90.0)  HOT_TEMPS = TEMPERATURES
WHERE (TEMPERATURES < 32.0)  COLD_TEMPS = TEMPERATURES
```


7.5.4.2 WHERE construct

The WHERE construct is defined as follows:

R739	*where_construct*	**is**	*where_construct_stmt* [*assignment_stmt*] ... [*elsewhere_stmt* [*assignment_stmt*] ...] *end_where_stmt*
R740	*where_construct_stmt*	**is**	WHERE (*mask_expr*)
R741	*mask_expr*	**is**	*logical_expr*
R742	*elsewhere_stmt*	**is**	ELSEWHERE
R743	*end_where_stmt*	**is**	END WHERE

The definition of the WHERE construct can be simplified to the following general format:

```
WHERE ( logical_expr )
  [ assignment_stmt ] ...
  [ ELSEWHERE
  [ assignment_stmt ] ... ]
END WHERE
```

In a WHERE construct, only the WHERE statement can be labeled as a branch target statement.

The WHERE *block* is the set of assignments between the WHERE construct statement and the ELSEWHERE statement (or END WHERE statement, if the ELSEWHERE statement is not present). The ELSEWHERE *block* is the set of assignment statements between the ELSEWHERE and the END WHERE statements.

Evaluation of the logical expression results in a logical array, which is treated as a mask. The mask array must conform with the variables being defined in all of the array assignment statements in the construct.

Each assignment in the WHERE block assigns a value to each array element that corresponds with a mask array element that is true.

Each assignment in the ELSEWHERE block assigns a value to each array element that corresponds with a mask array element that is false.

The ELSEWHERE block is optional; when it is not present, no assignment is made to elements corresponding to mask array elements that are false.

All of the assignment statements are executed in sequence as they appear in the construct (in both the WHERE and ELSEWHERE blocks).

Any elemental intrinsic operation or function within an array assignment statement is evaluated only for the selected elements. In the following example, the square roots are taken only of the elements of A that are positive:

```
REAL  A(10, 20)
   ...
WHERE (A > 0.0)
  SQRT_A = SQRT(A)
END WHERE
```

An elemental function reference is evaluated independently for each element, and only those elements needed in the array assignment are referenced. Elemental function references are only those intrinsic functions classified as elemental functions, such as ABS, INT, and COS.

Nonelemental function references, including defined operations, in the array assignment statements are completely evaluated, even though all elements of the resulting array may not be used.

```
REAL  A(2, 3), B(3, 10), C(2, 10), D(2, 10)
INTRINSIC  MATMUL
  ...
WHERE (D < 0.0)
  C = MATMUL(A, B)
END WHERE
```

The matrix product `A` multiplied by `B` is performed, yielding all elements of the product, and only for those elements of `D` that are negative are the assignments to the corresponding elements of `C` made.

7.5.4.3 Differences between the `WHERE` construct and control constructs

One major difference between the `WHERE` construct and control constructs has been described in subsection 7.5.4, page 318. Another difference is that no transfers out of `WHERE` or `ELSEWHERE` blocks are possible (except by a function reference) because only intrinsic assignment statements are permitted within these blocks. Note that the execution of statements in the `WHERE` block can affect variables referenced in the `ELSEWHERE` block (because the statements in both blocks are executed).

Controlling Execution [8]

A program performs its computation by executing the statements in sequence from beginning to end. Control constructs and branching statements modify this normal sequential execution of a program. The modification can select blocks of statements and constructs for execution or repetition, or can transfer control to another statement in the program.

As outlined in section 2, which begins on page 9, the statements and constructs making up a program are of two sorts, nonexecutable and executable. The *nonexecutable statements* establish the environment under which the program runs. The *executable statements* and *executable constructs* , some of which are *action statements,* perform computations, assign values, perform input/output (I/O) operations, or control the sequence in which the other executable statements and constructs are executed. This section describes the latter group of executable statements, the control statements and control constructs.

Control constructs and control statements alter the usual sequential execution order of statements and constructs in a program. This execution order is called the *normal execution sequence.* The control constructs are block constructs and consist of the `IF` construct, the `DO` construct, and the `CASE` construct. A nonblock form of the `DO` construct is also available. Individual statements that alter the normal execution sequence include the `CYCLE` and `EXIT` statements which are special statements for `DO` constructs, branch statements such as arithmetic `IF` statements, various forms of `GO TO` statements, and the statements that cause execution to cease such as the `STOP` and `PAUSE` statements.

With any of the block constructs, a construct name can be used to identify the construct and to identify which `DO` construct, particularly in a nest of `DO` constructs, is being terminated or cycled when using the `EXIT` or `CYCLE` statements.

The execution sequence

8.1

There is an established execution sequence for action statements in a Fortran program. Normally, a program or subprogram begins with the first executable statement in that program or subprogram and continues with the next executable statement in the order in which these statements appear . However, there are executable constructs and statements that cause statements to be executed in an order that is different from the order in which they appear in the program. These are either control constructs or branching statements.

There are two basic ways to affect the execution sequence. One is to use an executable construct that selects a block of statements and constructs for execution. The second is to execute a statement that branches to a specific statement in the program. In almost all cases, the use of constructs will result in programs that are more readable and maintainable, so constructs are discussed first, followed by branching statements.

Blocks and executable constructs

8.2

A *control construct* consists of one or more blocks of statements and constructs and the control logic that explicitly or implicitly encloses these blocks. Based on a control condition, a block of statements and constructs is selected for execution. A block is a sequence of zero or more statements and constructs, and it is defined as follows:

R801	*block*	**is**	[*execution_part_construct*] ...

A block of statements and constructs is treated as a whole. Either the block as a whole is executed or it is not executed. Whether or not the block is executed is determined by expressions in the control logic of the construct. Note that not every statement or construct in the block need be executed; for example, a branch statement early in the block can prevent subsequent statements in the block from being executed. This is still considered a complete execution of the block.

An executable construct consists of one or more blocks of statements surrounded by control statements. The construct usually contains an initial statement before a block and a terminal statement after the block. The construct includes conditions that determine which block in the construct is executed.

There are three executable constructs that contain blocks:

- `IF` construct
- `CASE` construct
- `DO` construct

The `IF` construct existed in FORTRAN 77. The `CASE` construct is new in Fortran 90. Extensions have been made to the FORTRAN 77 `DO` loop, which permit more flexible control of blocks.

There is also a construct called the `WHERE` construct that controls array assignment for individual elements (masked array assignment) as opposed to controlling flow of the program statements. Even though it looks like a control construct, it really is a construct for unconditional but masked array assignment. For information on masked array assignment, see subsection 7.5.4, page 318.

Construct names are described in the introductory material for blocks and also with each construct and statement that uses them. The name, if used, must appear on the same line as the initial statement of the construct and a matching name must appear on the terminal statement of the construct.

Some of the general rules and restrictions that apply to blocks and control of blocks are as follows:

- The statements of a block are executed in order unless there is a control construct or statement within the block that changes the sequential order.
- A block, as an integral unit, must be completely contained within a construct.
- A block can be empty; that is, it can contain no statements or constructs at all.

- A branching or control construct within a block that transfers to a statement or construct within the same block is permitted.
- Branching to a statement or construct within a block from outside the block is prohibited. (Even branching to the first executable statement within a block from outside the block is prohibited.)
- Exiting from a block can be done from anywhere within the block.
- References to procedures are permitted within a block.

`IF` construct and `IF` statement

8.3

An `IF` construct selects at most one block of statements and constructs within the construct for execution. The `IF` statement controls the execution of only one statement; in previous Fortran standards it was called the logical `IF` statement. The arithmetic `IF` statement is not the same as the `IF` statement; it is a branching statement that is designated as obsolescent.

The `IF` construct

8.3.1

The `IF` construct contains one or more executable blocks; at most one block is executed, and it is possible for no block to be executed when there is no `ELSE` statement.

8.3.1.1 Form of the IF construct

The IF construct is defined as follows:

```
R802   if_construct    is   if_then_stmt
                                    block
                                [ else_if_stmt
                                    block ] ...
                                [ else_stmt
                                    block ]
                            end_if_stmt

R803   if_then_stmt    is   [ if_construct_name : ] ≡
                            ≡ IF ( scalar_logical_expr ) THEN

R804   else_if_stmt    is   ELSE IF ( scalar_logical_expr ) ≡
                            ≡ THEN [ if_construct_name ]

R805   else_stmt       is   ELSE [ if_construct_name ]

R806   end_if_stmt     is   END IF [ if_construct_name ]
```

Branching to an ELSE IF or an ELSE statement is prohibited.

Branching to an END IF statement is allowed from any block within the IF construct. Branching to an END IF statement from outside the IF construct is allowed but is designated as an obsolescent feature.

If a construct name appears on the IF-THEN statement, the same name must appear on the corresponding END IF statement.

The construct names on the ELSE IF and ELSE statements are optional, but if present must be the same name as the one on the IF-THEN statement. If one such ELSE IF or ELSE statement has a construct name, the others are not required to have a construct name.

The same construct name must not be used for different named constructs in the same scoping unit; thus, two IF constructs must not be both named INNER in the same executable part, for example.

Section 8

8.3.1.2 Execution of the IF construct

The logical expressions are evaluated in order until one is found to be true. The block following the first true condition is executed, and the execution of the IF construct terminates. Subsequent true conditions in the construct have no effect. There may be no logical expressions found to be true in the construct. In this case, the block following the ELSE statement is executed if there is one; otherwise, no block in the construct is executed.

Figure 16 indicates the execution flow for an IF construct.

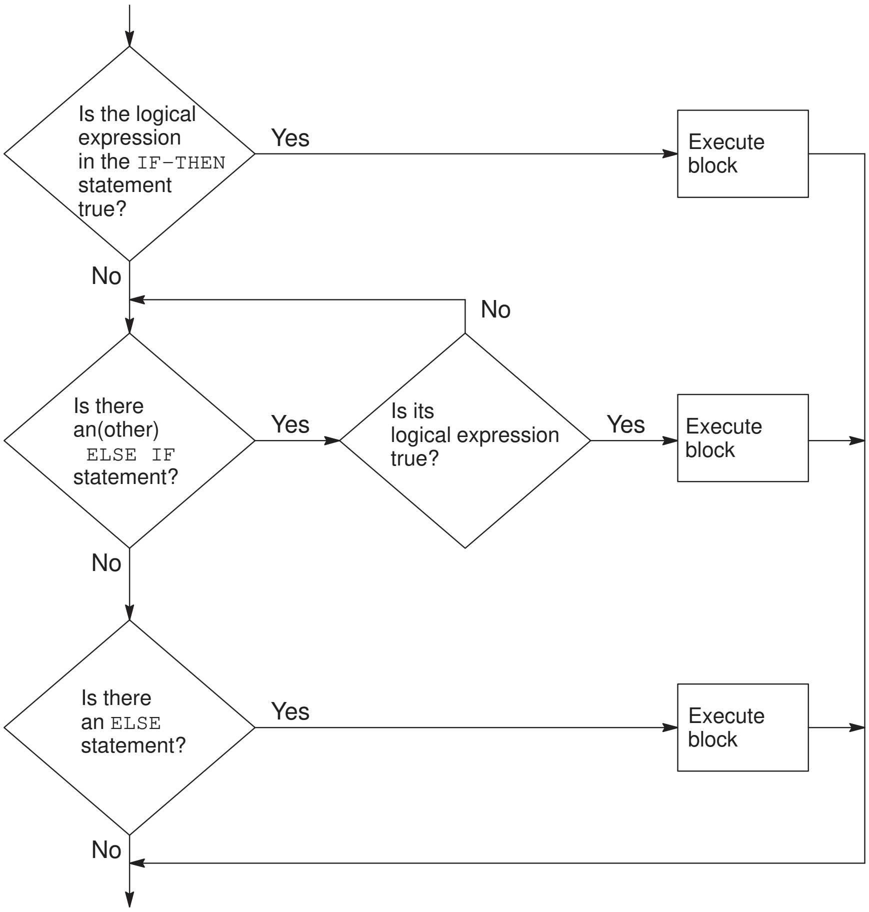

Figure 16. Execution flow for an `IF` construct

Section 8

Example:

```
IF (I < J) THEN
   X = Y + 5.0
ELSE IF (I > 100) THEN
   X = 0.0
   Y = -1.0
ELSE
   X = -1.0
   Y = 0.0
END IF
```

If `I` is less than `J`, the statement `X = Y + 5.0` is executed and execution proceeds following the `END IF` statement. If `I` ≥ `J` and if `I > 100`, the two statements following the `ELSE IF` statement are executed and execution proceeds following the `END IF` statement. If neither of these conditions is true, the block after the `ELSE` statement is executed.

Section 8

`IF` statement
8.3.2

The `IF` statement controls a single action statement, as opposed to a block of statements.

Form of the `IF` statement
8.3.2.1

The `IF` statement is defined as follows:

R807	*if_stmt*	**is**	`IF` (*scalar_logical_expr*) *action_stmt*

Example:

```
IF (S < T) S = 0.0
```

Execution of the `IF` statement
8.3.2.2

The scalar logical expression is evaluated. If true, the action statement is executed. If false, the action statement is not executed, and control passes to the next statement in the program.

The action statement must not be an `IF` statement or an `END` statement for a program, function, or subroutine. Note that the action statement cannot be any of the other `END` statements, such as `END DO` because they are not action statements.

If the logical expression contains a function reference, its evaluation may have side effects that modify the action statement.

A complete list of the action statements can be found in subsection 2.5, page 28. Action statements change the definition state of variables or the condition of the I/O system, or are control statements. Specification statements such as type declaration statements, `FORMAT` statements, and `ENTRY` statements are not action statements. Note that constructs are not action statements.

Section 8

`CASE` construct

8.4

The `CASE` construct is a new feature in Fortran 90. It, like the `IF` construct, consists of a number of blocks, of which at most one is selected for execution. The selection is based on the value of the scalar expression in the `SELECT CASE` statement at the beginning of the construct; the value of this expression is called the *case index.* The case selected is the one for which the case index matches a case selector value in a `CASE` statement. There is an optional default case that, in effect, matches all values not matched by any other `CASE` statement in the construct.

Form of the `CASE` construct

8.4.1

The general form of the `CASE` construct is as follows:

```
[ case_construct_name : ] SELECT CASE ( case_expression )
  [ CASE ( case_value_range_list ) [ case_construct_name ]
      block ] ...
  [ CASE DEFAULT [ case_construct_name ]
      block ]
END SELECT [ case_construct_name ]
```

The case construct is defined as follows:

R808	*case_construct*	**is**	*select_case_stmt* [*case_stmt* *block*] ... *end_select_stmt*
R809	*select_case_stmt*	**is**	[*case_construct_name* **:**] `SELECT CASE` (*case_expr*)
R810	*case_stmt*	**is**	`CASE` *case_selector* [*case_construct_name*]
R811	*end_select_stmt*	**is**	`END SELECT` [*case_construct_name*]
R812	*case_expr*	**is**	*scalar_int_expr*
		or	*scalar_char_expr*
		or	*scalar_logical_expr*
R808	*case_construct*	**is**	*select_case_stmt*
R813	*case_selector*	**is**	(*case_value_range_list*)
		or	`DEFAULT`
R814	*case_value_range*	**is**	*case_value*
		or	*case_value* **:**
		or	**:** *case_value*
		or	*case_value* **:** *case_value*
R815	*case_value*	**is**	*scalar_int_initialization_expr*
		or	*scalar_char_initialization_expr*
		or	*scalar_logical_initialization_expr*

The statement containing the keywords `SELECT CASE` is called the `SELECT CASE` *statement*. The statement beginning with the keyword `CASE` is called the `CASE` *statement*. The statement beginning with the keywords `END SELECT` is called the `END SELECT` *statement*. A case value range list enclosed in parenthesis or the `DEFAULT` keyword is called a *case selector.*

If a construct name is present on a `SELECT CASE` statement, it must also appear on the `END SELECT` statement.

Any of the case selector statements may or may not have a construct name. If one does, it must be the same name as the construct name on the `SELECT CASE` statement.

A `CASE` statement with the case selector `DEFAULT` is optional. If it is present, it is not required to be the last `CASE` statement.

The *case_expr* must be a scalar expression of type integer, character, or logical.

Within a particular `CASE` construct, the case expression and all case values must be of the same type and must have the same kind type parameter values. If the character type is used, different character lengths are allowed.

Each *case_value* must be a scalar initialization expression of the same type as the *case expression*. An *initialization expression* is an expression that can be evaluated at compile time; that is, essentially, a constant expression.

The colon forms of the case values expressing a range can be used for expressions in the construct of type integer and character but not type logical. For example, the following `CASE` statement would select all character strings that collate between `BOOK` and `DOG`, inclusive:

```
CASE ('BOOK':'DOG')
```

After expression evaluation, there must be no more than one case selector that matches the case index. In other words, overlapping case values and case ranges are prohibited.

Branching to the `END SELECT` statement is allowed only from within the construct. Branching to a `CASE` statement is prohibited; branching to the `SELECT CASE` statement is allowed, however.

The following example shows the CASE construct:

```
! Compute the area with a formula  appropriate for
! the shape of the object
FIND_AREA: &
   SELECT CASE (OBJECT)
     CASE (CIRCLE)
         AREA = PI * RADIUS ** 2
     CASE (SQUARE)
         AREA = SIDE * SIDE
     CASE (RECTANGLE)
         AREA = LENGTH * WIDTH
     CASE DEFAULT
        PRINT*, "Unable to compute area."
   END SELECT  FIND_AREA
```


Execution of the CASE construct

8.4.2

The case index (the scalar expression) in the SELECT CASE statement is evaluated in anticipation of matching one of the case values preceding the blocks. The case index must match at most one of the selector values. The block following the case matched is executed, the CASE construct terminates, and control passes to the next executable statement or construct following the END SELECT statement of the construct. If no match occurs and the CASE DEFAULT statement is present, the block after the CASE DEFAULT statement is selected. If there is no CASE DEFAULT statement, the CASE construct terminates, and the next executable statement or construct following the END SELECT statement of the construct is executed. If the case value is a single value, a match occurs if the index is equal to the case value (determined by the rules used in evaluating the equality or equivalence operator). If the case value is a range of values, there are three possibilities to determine a match depending on the form of the range:

Case value range	Condition for a match
$case\_value_1 : case\_value_2$	$case\_value_1 \le case\_index \le case\_value_2$
$case\_value$:	$case\_value \le case\_index$
: $case\_value$	$case\_value \ge case\_index$

Figure 17 illustrates the execution of a CASE construct.

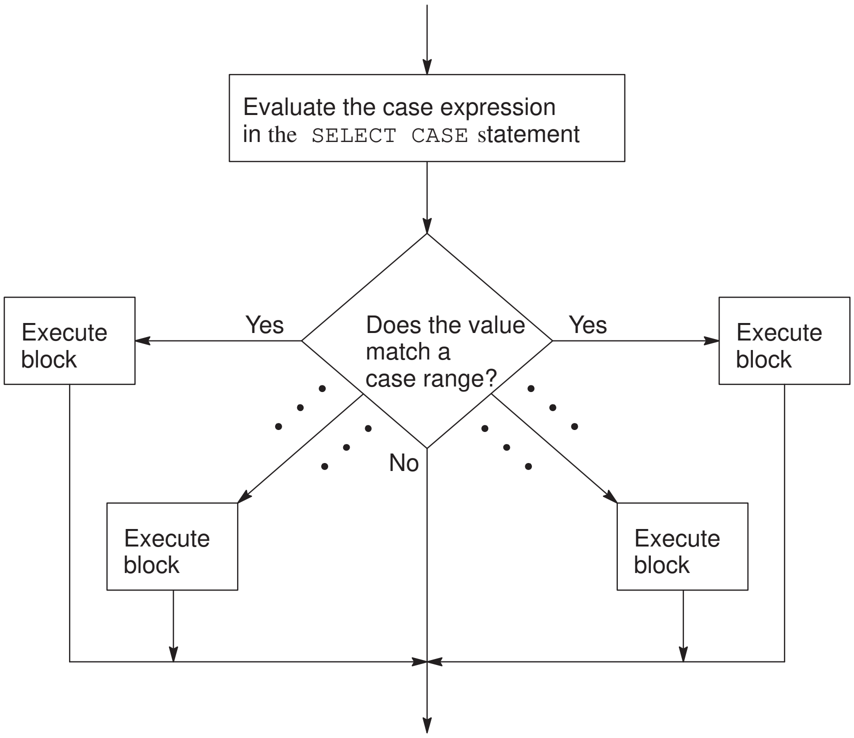

Figure 17. Execution flow for a CASE construct

Example 1:

```
INDEX = 2
SELECT CASE (INDEX)
  CASE (1)
     X = 1.0
  CASE (2)
     X = 2.0
  CASE DEFAULT
     X = 99.0
END SELECT
```

The case expression INDEX has the value 2. The block following the case value of 2 is executed; that is, the statement X = 2.0 is executed, and execution of the CASE construct terminates.

Example 2:

```
COLOR = 'GREEN'
SELECT CASE (COLOR)
  CASE ('RED')
     STOP
  CASE ('YELLOW')
     CALL  PROCEED_IF_YOU_CAN_SAFELY
  CASE ('GREEN')
     CALL  GO_AHEAD
END SELECT
```

This example uses selectors of type character. The expression COLOR has the value GREEN, and therefore the procedure GO_AHEAD is executed. When it returns, the execution of the CASE statement terminates, and the executable statement after the END SELECT statement executes next.

DO construct

8.5

The DO construct controls the number of times a sequence of statements and constructs within the range of a loop is executed. There are three steps in the execution of a DO construct:

1. If execution of the DO construct is controlled by a DO variable, the expressions representing the parameters that determine the number of times the range is to be executed are evaluated (step 1 of Figure 18).

2. A decision is made as to whether the range of the loop is to be executed (step 2 of Figure 18).

3. If appropriate, the range of the loop is executed (step 3a of Figure 18); the DO variable, if present, is updated (step 3b of Figure 18); and step 2 is repeated.

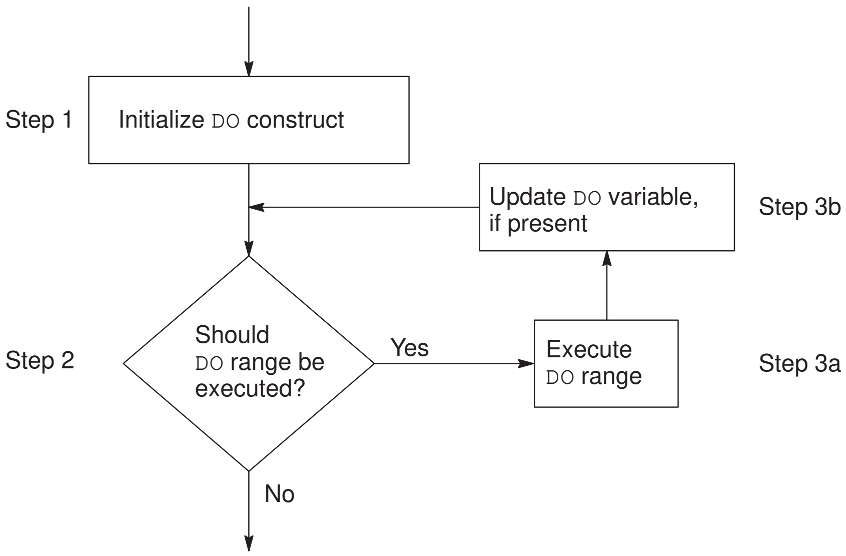

Figure 18. Execution flow for a DO construct

In FORTRAN 77, execution of a DO loop was controlled by a DO variable that was incremented a certain number of times as prescribed in the initial DO statement. In Fortran 90, this option remains available, but there are two additional ways of controlling the loop; one is the DO WHILE and the other is the simple DO, sometimes called DO forever.

There are two basic forms of the DO construct, the block DO and the nonblock DO.

Modern programming practice favors the block DO form, so the block DO form is the recommended construct. The nonblock DO form is obsolescent; it exists for compatibility with FORTRAN 77. The block DO contains all of the functionality of the nonblock DO and vice versa. Indeed, both forms of DO construct permit the DO WHILE and DO forever forms of loops. The feature distinguishing the two forms is that the block DO construct is always terminated by an END DO or CONTINUE statement whereas the nonblock DO construct either terminates with an action statement or shares a termination statement with another nonblock DO construct.

The following example shows a block DO construct:

```
DO I = 1, N
  SUM = SUM + A(I)
END DO
```

The following example shows a nonblock DO construct used to perform the same computation:

```
   DO 10 I = 1, N
10 SUM = SUM + A(I)
```

Form of the block DO construct
8.5.1

The block DO construct is a DO construct that terminates with an END DO statement or a CONTINUE statement that is not shared with another DO construct.

The block `DO` construct is defined as follows:

R817	*block_do_construct*	**is**	*do_stmt* *do_block* *end_do*
R818	*do_stmt*	**is**	*label_do_stmt*
		or	*nonlabel_do_stmt*
R819	*label_do_stmt*	**is**	[*do_construct_name* :] `DO` *label* [*loop_control*]
R820	*nonlabel_do_stmt*	**is**	[*do_construct_name* :] `DO` [*loop_control*]
R821	*loop_control*	**is**	[`,`] *do_variable* = *scalar_numeric_expr* `,` ≡ ≡ *scalar_numeric_expr* [`,` *scalar_numeric_expr*]
		or	[`,`] `WHILE (` *scalar_logical_expr* `)`
R822	*do_variable*	**is**	*scalar_variable*
R823	*do_block*	**is**	*block*
R824	*end_do*	**is**	*end_do_stmt*
		or	*continue_stmt*
R825	*end_do_stmt*	**is**	`END DO` [*do_construct_name*]

The `DO` variable must be a scalar named variable of type integer, default real, or double-precision real. (This excludes variables that are array elements, structures, and components of structures.) Using a real or double-precision `DO` variable is obsolescent.

Each scalar numeric expression in the loop control must be of type integer, default real, or double-precision real. Using numeric expressions of type real or double-precision real for the `DO` loop parameters is obsolescent.

If the `DO` statement of a block `DO` construct has a construct name, the corresponding *end_do* must be an `END DO` statement that has the same construct name. If the `DO` statement of a block `DO` construct does not have a construct name, the corresponding *end_do* must not have a construct name.

If the DO statement does not contain a label, the corresponding *end_do* must be an END DO statement. If the DO statement does contain a label, the corresponding *end_do* must be identified with the same label. By definition, a block DO construct cannot share its terminal statement with another DO construct, even if it is a labeled statement. If a DO construct does share its terminal statement with another DO construct, it is a nonblock DO construct. Refer to the following examples:

```
SUM = 0.0
DO I = 1, N
  SUM = SUM + X(I) ** 2
END DO

FOUND = .FALSE.
I = 0
DO WHILE (.NOT. FOUND .AND. I < LIMIT )
  IF (KEY == X(I))  THEN
    FOUND = .TRUE.
  ELSE
    I = I + 1
  END IF
END DO

NUM_ITERS = 0
DO
  ! F and F_PRIME are functions
  X1 = X0 - F(X0) / F_PRIME(X0)
  IF (ABS(X1-X0) < SPACING(X0) .OR. &
      NUM_ITERS > MAX_ITERS)  EXIT
  X0 = X1
  NUM_ITERS = NUM_ITERS + 1
END DO

INNER_PROD = 0.0
DO 10 I = 1, 10
   INNER_PROD = INNER_PROD + X(I) * Y(I)
10 CONTINUE

LOOP: DO I = 1, N
        Y(I) = A * X(I) + Y(I)
      END DO LOOP
```

Section 8

Although a DO construct can have both a label and a construct name, use of both is not in the spirit of modern programming practice where the use of labels is minimized.

8.5.2 Form of the nonblock DO construct

The obsolescent nonblock DO construct is a DO construct that either shares a terminal statement with another DO construct, or the terminal statement is an action statement. The nonblock DO construct always uses a label to specify the terminal statement of the construct. The nonblock DO construct that ends with an action statement is defined as follows:

R826	*nonblock_do_construct*	**is**	*action_term_do_construct*
		or	*outer_shared_do_construct*
R827	*action_term_do_construct*	**is**	*label_do_stmt* *do_body* *do_term_action_stmt*
R828	*do_body*	**is**	[*execution_part_construct*] ...
R829	*do_term_action_stmt*	**is**	*action_stmt*

The nonblock DO construct that shares a termination statement is defined as follows:

R826	*nonblock_do_construct*	**is**	*action_term_do_construct*
		or	*outer_shared_do_construct*
R830	*outer_shared_do_construct*	**is**	*label_do_stmt* *do_body* *shared_term_do_construct*
R831	*shared_term_do_construct*	**is**	*outer_shared_do_construct*
		or	*inner_shared_do_construct*
R832	*inner_shared_do_construct*	**is**	*label_do_stmt* *do_body* *do_term_shared_stmt*
R833	*do_term_shared_stmt*	**is**	*action_stmt*

The last statement in a nonblock DO construct (the statement in which the loop label is defined), is called the DO *termination* or *terminal statement* of that construct.

An *action_term_do_construct* is a nonblock DO construct that does not share its DO termination with any other nonblock DO construct. An *outer_shared_do_construct* is a nonblock DO construct that shares its DO termination with at least one inner nonblock DO construct.

The DO termination of an *action_term_do_construct* must not be one of the following:

- A GO TO statement
- A RETURN statement
- A STOP statement
- An EXIT statement
- A CYCLE statement

- An END statement for a program or subprogram
- An arithmetic IF statement
- An assigned GO TO statement

Note that a *do_term_action_stmt* is an *action_stmt*. A CONTINUE statement is an *action_stmt*, but by definition, if a DO construct ends with a CONTINUE statement, it is a block DO construct. Also note that a *do_term_action_stmt* cannot be any kind of END statement; END statements other than program or subprogram END statements are not specifically named in the preceding list because they are not *action_stmt* s.

The DO termination must be identified with a label and the corresponding DO statement must refer to the same label.

The DO termination of an *outer_shared_do_construct* must not be a GO TO statement, a RETURN statement, a STOP statement, an EXIT statement, a CYCLE statement, an END statement for a program or subprogram, an arithmetic IF statement, or an assigned GO TO statement. Note that DO termination cannot be any other END statement because the other END statements are not *action_stmt* s.

The DO termination must be identified with a label and all DO statements of the shared termination DO construct must refer to the same label.

The following are examples of DO constructs that are nonblock DO constructs because the DO terminations are action statements.

```
   PROD = 1.0
   DO 10 I = 1, N
10   PROD = PROD * P(I)

   FOUND = .FALSE.
   I = 0
   DO 10 WHILE (.NOT. FOUND .AND. I < LIMIT)
     I = I + 1
10   FOUND = KEY == X(I)
```

Section 8

The following are examples of DO constructs that are nonblock DO constructs because the DO terminations are shared.

```
      DO 10 I = 1, N
        DO 10 J = 1, N
10        HILBERT(I, J) = 1.0 / REAL(I + J)

      DO 20 I = 1, N
        DO 20 J = I+1, N
          T = A(I, J); A(I, J) = A(J, I); A(J, I) = T
20    CONTINUE
```

Range of a DO construct

8.5.3

The range of a DO construct consists of all statements and constructs following the DO statement, bounded by and including the terminal statement. The DO range can contain constructs, such as an IF construct, a CASE construct, or another DO construct, but the inner construct or constructs must be entirely enclosed within the nearest outer construct. If the range of a DO construct contains another construct, the constructs are said to be *nested.*

A branch to a statement within the range of a DO construct is diagnosed by the compiler as being unsafe.

ANSI/ISO: The Fortran 90 standard prohibits a branch into the range of a DO construct.

Active and inactive DO constructs

8.5.4

A DO construct is either active or inactive. A DO construct becomes *active* when the DO statement is executed. A DO construct becomes *inactive* when any one of the following situations occurs:

- The iteration count is zero at the time it is tested
- The WHILE condition is false at the time it is tested
- An EXIT statement is executed that causes an exit from the DO construct or any DO construct containing the DO construct

- A `CYCLE` statement is executed that causes cycling of any `DO` construct containing the `DO` construct
- There is a transfer of control out of the `DO` construct
- A `RETURN` statement in the `DO` construct is executed
- The program terminates for any reason

Execution of DO constructs

8.5.5

There are three forms of `DO` constructs, each with its own rules for execution: a `DO` construct with an iteration count, a `DO WHILE` construct, and a simple `DO` construct.

Section 8

DO construct with an iteration count

8.5.5.1

In this case, an iteration count controls the number of times the range of the loop is executed.

The general form of a `DO` statement using an iteration count is as follows:

`DO [` *label* `] [ , ]` *do_variable* `=` *start_expr* `,` *end_expr* `[ ,` *inc_expr* `]`

The `DO` variable and the expressions may be of type integer, default real, or double precision, but the use of default real or double precision is obsolescent. The following are examples of the iterative `DO` statement:

```
DO 10 I = 1, N
DO, J = -N, N
DO K = N, 1, -1
```

The iteration count
8.5.5.1.1

An iteration count is established for controlling the number of times the program executes the range of the DO construct. This is done by evaluating the expressions *start_expr* , *end_expr* , and *inc_expr* , and converting these values to the type of the DO variable. For example, let m_1, m_2, and m_3 be the values obtained:

- m_1 is the initial value of the DO variable
- m_2 is the terminal value the DO variable may assume
- m_3 is an optional parameter, specifying the DO variable increment

The value of m_3 must not be zero. If *expression* $_3$ is not present, m_3 is given the value 1. The iteration count is calculated from the following formula:

$$\text{MAX (INT ((}m_2 - m_1 + m_3\text{) / }m_3\text{), 0)}$$

Note that the iteration count is 0 if one of the following conditions is true:

- $m_1 > m_2$ and $m_3 > 0$
- $m_1 < m_2$ and $m_3 < 0$

Section 8

Controlling execution of the range of the DO construct
8.5.5.1.2

The steps that control the execution of the range of the DO construct are as follows:

1. The DO variable is set to m_1, the initial parameter (step 1 of Figure 18, page 337).
2. The iteration count is tested (step 2 of Figure 18, page 337). If it is 0, the DO construct terminates.
3. If the iteration count is not 0, the range of the DO construct is executed (step 3a of Figure 18, page 337). The iteration count is decremented by 1, and the DO variable is incremented by m_3 (step 3b of Figure 18, page 337). Steps 2 and 3 are repeated until the iteration count is 0.

After termination, the DO variable retains its last value, the one that it had when the iteration count was tested and found to be 0.

The DO variable must not be redefined or become undefined during the execution of the range of the DO construct. Note that changing the variables used in the expressions for the loop parameters during the execution of the DO construct does not change the iteration count; it is fixed when execution of the DO construct starts.

```
   N = 10
   SUM = 0.0
   DO 2 I = 1, N
      SUM = SUM + X(I)
      N = N + 1
 2 CONTINUE
```

The loop is executed 10 times; after execution I = 11 and N = 20.

```
 X = 20.
 DO I = 1, 2
    DO J = 1, 5
    X = X + 1.0
 END DO
 END DO
```

The inner loop is executed 10 times. After completion of the outer DO construct, J = 6, I = 3, X = 30.

If the second DO statement had been the following, the inner DO construct would not have executed at all; X would remain equal to 20; J would equal 5, its initial value; and I would be equal to 3:

```
 DO J = 5, 1
```

Section 8

DO WHILE construct
8.5.5.2

The DO WHILE form of the DO construct provides the ability to repeat the DO range while a specified condition remains true.

The general form of the DO WHILE statement is as follows:

DO [*label*] [,] WHILE (*expression*)

The following examples show the `DO WHILE` statement:

```
DO WHILE( K >= 4 )
DO 20 WHILE( .NOT. FOUND )
```

The `DO` range is executed repeatedly. Prior to each execution of the `DO` range, the logical expression is evaluated. If it is true, the range is executed; if it is false, the `DO WHILE` construct terminates.

```
SUM = 0.0
I = 0
DO WHILE (I < 5)
   I = I + 1
   SUM = SUM + I
END DO
```

The loop would execute five times, after which `SUM = 15.0` and `I = 5`.

Section 8

Simple `DO` construct
8.5.5.3

A `DO` construct without any loop control provides the ability to repeat statements in the `DO` range until the `DO` construct is terminated explicitly by some statement within the range. When the end of the `DO` range is reached, the first executable statement of the `DO` range is executed next.

The form of the simple `DO` statement is as follows:

`DO` [*label*]

Example:

```
DO
   READ *, DATA
   IF (DATA < 0) STOP
   CALL PROCESS (DATA)
END DO
```

The DO range executes repeatedly until a negative value of DATA is read, at which time the DO construct (and the program, in this case) terminates. The previous example, rewritten using a label, appears as follows:

```
      DO 100
         READ *, DATA
         IF (DATA <0) STOP
         CALL PROCESS(DATA)
  100 CONTINUE
```

Altering the execution sequence within the range of a DO construct
8.5.6

There are two statements that can appear only in the range of a DO construct that alter the execution sequence of the DO construct. One is the EXIT statement; the other is the CYCLE statement. Other statements, such as branch statements, the RETURN statement, and the STOP statement, also alter the execution sequence but are not restricted to DO constructs as are the EXIT and CYCLE statements.

EXIT statement
8.5.6.1

The EXIT statement immediately causes termination of the DO construct. No further action statements within the range of the DO construct are executed. It can appear in either the block or nonblock form of the DO construct, except that it must not be the DO termination of the nonblock form.

The EXIT statement is defined as follows:

R835	*exit_stmt*	**is**	EXIT [*do_construct_name*]

If the EXIT statement has a construct name, it must be within the DO construct with the same name; when it is executed, the named DO construct is terminated as well as any DO constructs containing the EXIT statement and contained within the named DO construct.

If the EXIT statement does not have a construct name, the innermost DO construct in which the EXIT statement appears is terminated.

Example 1: In the following example, the DO construct has a construct name, LOOP_8; the DO range is executed repeatedly until the condition in the IF statement is met, when the DO construct terminates:

```
LOOP_8 : DO
   ...
   IF (TEMP == INDEX) EXIT LOOP_8
   ...
END DO LOOP_8
```

Example 2: In the following example, when the EXIT statement in the IF statement is executed, both the inner loop and the outer loop are terminated:

```
OUTER_LOOP:  DO I = 1, 10
INNER_LOOP:  DO J = 1, 10
               ...
               IF (TEMP == INDEX) EXIT OUTER_LOOP
               ...
               END DO INNER_LOOP
               ...
             END DO OUTER_LOOP
```


CYCLE statement
8.5.6.2

In contrast to the EXIT statement, which terminates execution of the DO construct entirely, the CYCLE statement interrupts the execution of the DO range and begins a new cycle of the DO construct, with appropriate adjustments made to the iteration count and DO variable, if present. It can appear in either the block or nonblock form of the DO construct, except it must not be the DO termination of the nonblock form. When the CYCLE statement is executed in the nonblock form, the DO termination is not executed.

The CYCLE statement is defined as follows:

R834	*cycle_stmt*	**is**	CYCLE [*do_construct_name*]

If the `CYCLE` statement has a construct name, it must be within the `DO` construct with the same name; when it is executed, the execution of the named `DO` construct is interrupted, and any `DO` construct containing the `CYCLE` statement and contained within the named `DO` construct is terminated.

If the `CYCLE` statement does not have a construct name, the innermost `DO` construct in which the `CYCLE` statement appears is interrupted.

The `CYCLE` statement can be used with any form of the `DO` statement and causes the next iteration of the `DO` range to begin, if permitted by the condition controlling the loop.

Upon interruption of the `DO` construct, if there is a `DO` variable, it is updated and the iteration count is decremented by `1`. Then, in all cases, the processing of the next iteration begins.

In the following example, the loop is executed as long as `INDEX` is nonnegative. If `INDEX` is negative, the loop is terminated. If `INDEX` is `0`, the latter part of the loop is skipped.

```
DO
   . . .
   INDEX = . . .
   . . .
   IF (INDEX < 0) EXIT
   IF (INDEX == 0) CYCLE
   . . .
END DO
```


Branching

8.6

Branching is a transfer of control from the current statement to another statement or construct in the program unit. A branch alters the execution sequence. This means that the statement or construct immediately following the branch is usually not executed. Instead, some other statement or construct is executed, and the execution sequence proceeds from that point. The terms *branch statement* and *branch target statement* are used to distinguish between the transfer statement and the statement to which the transfer is made.

An example of branching is provided by the `GO TO` statement. It is used to transfer control to a statement in the execution sequence that is usually not the next statement in the program, although this is not prohibited.

The statements that can be branch target statements are those classified as action statements plus the `IF-THEN` statement, `SELECT CASE` statement, `DO` statement, `WHERE` statement, and a few additional statements in limited situations.

The additional statements that can be branch targets in limited contexts are as follows:

- An `END SELECT` statement, provided the branch is taken from within the `CASE` construct.
- A `DO` termination, provided the branch is taken from within the `DO` construct.
- An `END IF` statement, provided that the branch is taken from within the `IF` construct. A branch to an `END IF` statement from outside the `IF` construct is allowed, but this use is designated as obsolescent.

Section 8

The standard does not permit a branch to a statement within a block from outside the block. The CF90 compiler, however, permits these branches for compatibility with the CF77 compiling system; such branches are diagnosed as being unsafe.

> **ANSI/ISO:** The Fortran 90 standard does not permit branches into executable blocks.

8.6.1 Use of labels in branching

A statement label is a means of identifying the branch target statement. Any statement in a Fortran program can have a label. However, if a branch statement refers to a statement label, some statement in the program unit must have that label, and the statement label must be on an allowed branch target statement.

As described in subsection 3.2.5, page 57, a label is a string of from one to five decimal digits; leading zeros are not significant. Note that labels can be used in both free and fixed source forms.

GO TO statement
8.6.2

The GO TO statement is an unconditional branching statement that alters the execution sequence.

Form of the GO TO statement
8.6.2.1

The GO TO statement is defined as follows:

R836	*goto_stmt*	**is**	GO TO *label*

The label must be a branch target statement in the same scoping unit as the GO TO statement (that is, in the same program unit, excluding labels on statements in internal procedures, derived-type definitions, and interface blocks).

Execution of the GO TO statement
8.6.2.2

When the GO TO statement is executed, the next statement that is executed is the branch target statement identified with the label specified. Execution proceeds from that point.

```
GO TO 200    ! This is an unconditional branch and
             ! always goes to 200.
             !
X = 1.0      ! Because this statement follows a GO
             ! TO statement and is unlabeled, it is
             ! not reachable.
GO TO 10
GO TO 010    ! 10 and 010 are the same label.
```

Computed GO TO statement
8.6.3

The computed GO TO statement transfers to one of a set of branch target statements based on the value of an integer expression, selecting the branch target from a list of labels. The CASE construct provides a similar functionality in a more structured form.

8.6.3.1 Form of the computed GO TO statement

The computed `GO TO` statement is defined as follows:

R837	*computed_goto_stmt*	**is**	`GO TO (` *label_list* `) [ , ]` *scalar_int_expr*

If there are n labels in the list and the expression has one of the values from 1 to n, the value identifies a statement label in the list: the first, second, ... , or nth label. A branch to the statement with that label is executed.

If the value of the expression is less than 1 or greater than n, no branching occurs and execution continues with the next executable statement or construct following the computed `GO TO` statement.

Each label in the list must be the label of a branch target statement in the same scoping unit as the computed `GO TO` statement.

A label can appear more than once in the list of target labels.

```
GO TO ( 10, 20 ), SWITCH
GO TO ( 100, 200, 3, 33 ),  2*I-J
```

In the following example, if `SWITCH` has the value 1 or 3, the assignment statement labeled `10` is executed; if it has the value 2, the assignment statement labeled `11` is executed. If it has a value less than 1 or greater than 3, the assignment statement `Y = Z` is executed, because it is the next statement after the computed `GO TO` statement, and the statement with label `10` is executed next.

```
   SWITCH = . . .
   GO TO (10, 11, 10) SWITCH
   Y = Z
10 X = Y + 2.
   . . .
11 X = Y
```

`CONTINUE` statement
8.6.4

The `CONTINUE` statement is defined as follows:

R841	*continue_stmt*	**is**	`CONTINUE`

Typically , the statement has a label and is used for `DO` termination; however, it can serve as some other place holder in the program or as a branch target statement. It can appear without a label. The statement by itself does nothing and has no effect on the execution sequence or on program results.

```
100 CONTINUE
CONTINUE
```


`STOP` statement
8.6.5

The `STOP` statement terminates the program whenever and wherever it is executed. The `STOP` statement is defined as follows:

R842	*stop_stmt*	**is**	`STOP` [*stop_code*]
R843	*stop_code*	**is**	*scalar_char_constant*
CRI		**or**	*digit* . . . [*digit*]

The character constant or list of digits identifying the `STOP` statement is optional and is called a *stop code.*

When the *stop_code* is a string of digits, leading zeros are not significant; `10` and `010` are the same *stop_code* . You can specify from 1 to 80 *digit* s.

The stop code is accessible following program termination; the CF90 compiler sends it to the standard error file (`stderr`).

```
STOP
STOP 'Error #823'
STOP 20
```

ANSI/ISO: The Fortran 90 standard specifies from 1 to 5 *digit* s in the *stop_code* .

Obsolescent control statements

8.7

Three control facilities are designated as obsolescent. Their use is discouraged because there are other language constructs that reflect more modern programming practices. These statements are the `ASSIGN` and the assigned `GO TO` statements, the arithmetic `IF` statement, and the `PAUSE` statement.

`ASSIGN` and assigned `GO TO` statements

8.7.1

The `ASSIGN` statement assigns a statement label to an integer variable. During program execution, the variable can be assigned labels of branch target statements, providing a dynamic branching capability in a program. The unsatisfactory property of these statements is that the integer variable name can be used to hold both a label and an ordinary integer value, leading to errors that can be hard to discover and programs that could be difficult to read.

A frequent use of the `ASSIGN` statement and assigned `GO TO` statement is to simulate internal procedures, using the `ASSIGN` statement to record the return point after a reusable block of code has completed. The internal procedure mechanism of Fortran 90 provides this capability.

A second use of the `ASSIGN` statement is to simulate dynamic format specifications by assigning labels corresponding to different format statements to an integer variable and using this variable in I/O statements as a format specifier. This use can be accomplished in a clearer way by using character strings as format specifications. Thus, it is no longer necessary to use either the `ASSIGN` statement or the assigned `GO TO` statement.

8.7.1.1 Form of the ASSIGN and assigned GO TO statements

Execution of an ASSIGN statement assigns a label to an integer variable. Subsequently , this value can be used by an assigned GO TO statement or by an I/O statement to reference a FORMAT statement. The ASSIGN statement is defined as follows:

R838	*assign_stmt*	**is**	ASSIGN *label* TO *scalar_int_variable*

The variable must be a named variable of default integer type. It must not be an array element, an integer component of a structure, or an object of nondefault integer type.

The label must be the label of a branch target statement or the label of a FORMAT statement in the same scoping unit as the ASSIGN statement.

When defined with an integer value, the integer variable cannot be used as a label.

When assigned a label, the integer variable cannot be used as anything other than a label.

When the integer variable is used in an assigned GO TO statement, it must be assigned a label.

The variable can be redefined during program execution with either another label or an integer value.

```
ASSIGN 100 TO K
```

Execution of the assigned GO TO statement causes a transfer of control to the branch target statement with the label that had previously been assigned to the integer variable.

The assigned GO TO statement is defined as follows:

R839	*assigned_goto_stmt*	**is**	GO TO *scalar_int_variable* [[,] (*label_list*)]

Section 8

The variable must be a named variable of default integer type. That is, it must not be an array element, a component of a structure, or an object of nondefault integer type.

The variable must be assigned the label of a branch target statement in the same scoping unit as the assigned `GO TO` statement.

If the label list appears, the variable must have been assigned a label value that is in the list.

```
GO TO K
GO TO K (10, 20, 100)
```

8.7.2 Arithmetic `IF` statement

The arithmetic `IF` statement is a three-way branching statement based on an arithmetic expression.

The arithmetic `IF` statement is defined as follows:

R840	*arithmetic_if_stmt*	**is**	`IF (` *scalar_numeric_expr* `)` *label* `,` *label* `,` *label*

The same label can appear more than once in an arithmetic `IF` statement.

The numeric expression must not be of type complex.

Each statement label must be the label of a branch target statement in the same scoping unit as the arithmetic `IF` statement itself.

The execution begins with the evaluation of the expression. If the expression is negative, the branch is to the first label; if zero, to the second label; and if positive, to the third label.

The following example shows an arithmetic `IF` statement:

```
      READ *, DATA
      IF(DATA) 10, 20, 30
10    PRINT *, 'NEGATIVE VALUE'
      ...
20    PRINT *, 'ZERO VALUE'
      ...
30    PRINT *, 'POSTIVE VALUE'
      ...
```

8.7.3 `PAUSE` statement

The execution of the `PAUSE` statement suspends the execution of a program. This is now redundant, because a `WRITE` statement can be used to send a message to any device, and a `READ` statement can be used to wait for and receive a message from the same device.

The `PAUSE` statement is defined as follows:

R844	*pause_stmt*	**is**	`PAUSE` [*stop_code*]

The character constant or list of digits identifying the `PAUSE` statement is called the *stop_code* because it follows the same rules as those for the `STOP` statement' s stop code. The stop code is accessible following program suspension; the CF90 compiler sends the *stop_code* to the standard error file (`stderr`).

```
PAUSE
PAUSE 'Wait #823'
PAUSE 100
```

Section 8

Input and Output (I/O) Processing [9]

Many programs need data to begin a calculation. After the calculation is completed, often the results need to be printed, displayed graphically, or saved for later use. During execution of a program, sometimes there is a large amount of data produced by one part of the program that needs to be saved for use by another part of the program, and the amount of data is too large to store in variables, such as arrays. Also, the editing capabilities of the data transfer statements for internal files can be used for processing character strings. Each of these tasks is accomplished using Fortran I/O statements described in this section.

The I/O statements are as follows:

```
READ
PRINT
WRITE
OPEN
CLOSE
INQUIRE
BACKSPACE
ENDFILE
REWIND
```

The `READ` statement is a *data transfer input statement* and provides a means for transferring data from an external media to internal storage or from an internal file to internal storage through a process called *reading*. The `WRITE` and `PRINT` statements are both *data transfer output statements* and provide a means for transferring data from internal storage to an external media or from internal storage to an internal file. This process is called *writing*. The `OPEN` and `CLOSE` statements are both *file connection statements*. The `INQUIRE` statement is a *file inquiry statement*. The `BACKSPACE`, `ENDFILE`, and `REWIND` statements are *file positioning statements*.

The first part of this section discusses terms and concepts you need to gain a thorough understanding of all of the I/O facilities. These include internal and external files, formatted and unformatted records, sequential and direct access methods for files, advancing and nonadvancing I/O for the sequential formatted access method, file and record positions, units, and file connection properties. Following the concepts are descriptions of the `READ`, `WRITE`, and `PRINT` data transfer statements and the effect of these statements when they are executed. A model for the execution of data transfer statements and a description of the possible error and other conditions created during the execution of data transfer statements are provided next. Following the model are the descriptions of the `OPEN`, `CLOSE`, and `INQUIRE` statements that establish respectively the connection properties between units and files, that disconnect units and files, and that permit inquiry about the state of the connection between a unit and file. Lastly, file position statements are specified, which include the `BACKSPACE` and `REWIND` statements, followed by the description of the `ENDFILE` statement that creates end-of-file records.

The section concludes with a summary of the terms, concepts, and statements used for input and output processing and some examples.

Records, files, access methods, and units

9.1

Collections of data are stored in files. The data in a file is organized into records. Fortran treats a record, for example, as a line on a computer terminal, a line on a listing, or a logical record on a magnetic tape or disk file. However, the general properties of files and records do not depend on how the properties are acquired or how the files and records are stored. This subsection discusses the properties of records and files, and the various kinds of data transfer.

A *file* is a sequence of records. The following subsection describes the properties of records.

Cray Research physical files and records are described in the *UNICOS File Formats and Special Files Reference Manual*, publication SR–2014. If you are using the CF90 compiler on a SPARC system, you can assume that standard UNIX file formats are used.

Records

9.1.1

There are two kinds of records: data and end-of-file. A *data record* is a sequence of values.

The values in a data record can be represented in one of two ways: formatted or unformatted. *Formatted* data consists of characters that are viewable on some medium. For example, a record could contain the following four character values:

```
6
,
1
1
```

These are intended to represent the two numbers 6 and 11. In this case, the record might be represented schematically as shown in Figure 19.

6	,	1	1

Figure 19. A formatted record with four character values

Unformatted data consists of values represented just as they are stored in computer memory. For example, if integers are stored using a binary representation, an unformatted record, consisting of two integer values, 6 and 11, might look like Figure 20.

000000110	000001011

Figure 20. An unformatted record with two integer values

The values in a data record are either all formatted or all unformatted. A *formatted record* is one that contains only formatted data. It can be created by a person typing at a terminal or by a Fortran program that converts values stored internally into character strings that form readable representations of those values. When a program reads formatted data, the characters must be converted to the computer's internal representation of values. Even character values can be converted from one character representation in the record to another internal representation. The length of a formatted record is the number of characters in it; the length can be zero.

An *unformatted record* is one that contains only unformatted data. Unformatted records usually are created by running a Fortran program, but they can be created by other means. Unformatted data often requires less space on an external device. Also, it is usually faster to read and write because no conversion is required. However, it is not as suitable for reading by people and usually it is not suitable for transferring data from one computer to another because the internal representation of values is machine-dependent. The length of an unformatted data record depends on the number of values in it, but is measured in bytes; it can be zero. The length of an unformatted record that will be produced by a particular output list can be determined with the `INQUIRE` statement.

In general, a formatted record is read and written by a formatted data transfer I/O statement, and an unformatted record is read and written by an unformatted data transfer I/O statement.

Another kind of record is the *end-of-file record*; it has no value and has no length.

On Cray PVP systems and CRAY T3E systems, certain file structures allow more than one end-of-file record in a file. The COS blocking format is the default file structure for all UNICOS Fortran sequential unformatted files (except tape files). It can have multiple end-of-file records. For general information on records supported on Cray PVP systems, see the *Application Programmer's I/O Guide*, publication SG–2168.

For Cray Research structures other than COS blocked, and for structures on SPARC systems, an end-of-file record must be the last record of a file. It is used to mark the end of a file. To write it explicitly for files connected for sequential access, use the `ENDFILE` statement; to write it implicitly, use a file positioning statement (`REWIND` or `BACKSPACE` statement), close the file (`CLOSE` statement), open the file (`OPEN` statement), or use normal termination of a program.

ANSI/ISO: The Fortran 90 standard does not allow for multiple end-of-file records in a file.

Kinds of files
9.1.2

A file can contain a mixture of formatted and unformatted records.

> **ANSI/ISO:** The Fortran 90 standard does not allow formatted and unformatted records to be mixed in a file.

The file can be named. The length of the file name and path name depend on the platform, as follows:

- Cray PVP systems and CRAY T3E systems allow 256 characters for a file name. This file name can include a directory name and 1023 characters for the full path name.
- SPARC systems allow 255 characters for a file name. The full path name is limited to 1023 characters.

A distinction is made between files that are located on an external device like a disk, and files in memory accessible to the program. These two kinds of files are as follows:

- External files
- Internal files

The use of these files is illustrated schematically in Figure 21.

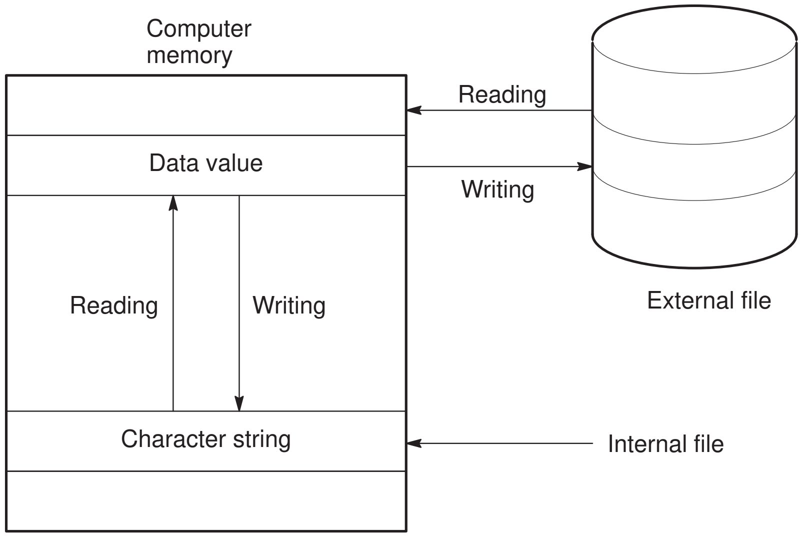

Figure 21. Internal and external files

External files
9.1.2.1

External files are located on external devices such as tapes, disks, or computer terminals. For each external file, there is a set of allowed access methods, a set of allowed forms, a set of allowed actions, and a set of allowed record lengths. These characteristics are determined by a combination of requests by the user of the file and by actions of the operating system. Each characteristic is discussed later in this section.

An external file connected to a unit has the position property. The file is positioned at the current record. In some cases, the file is positioned within the current record.

Internal files
9.1.2.2

The contents of internal files are stored as values of variables of type character . You can create the character values by using all the usual means of assigning character values, or you can create them with an output statement that specifies the variable as an internal file. Data transfer to and from internal files is described in subsection 9.2.9, page 404. Such data transfer to and from an internal file must use formatted sequential access I/O statements, including list-directed data transfer, but not namelist data transfer .

File connection, file positioning, and file inquiry must not be used with internal files. If the variable representing the internal file is a scalar, the file has just one record; if the variable is an array, the file has one record for each element of the array. The order of the records is the order of the elements in the array. The length of each record is the length of one array element.

File existence
9.1.2.3

Certain files are made known to the system for any executing program, and these files are said to *exist* at the time the program begins executing. A file might not exist because it is not anywhere on the disks accessible to a system or because the user of the program is not authorized to access the file.

In addition to files that are made available to programs for input, output, and other special purposes, programs can create files needed during and after program execution. When the program creates a file, it is said to exist, even if no data has been written into it. A file no longer exists after it has been deleted. Any of the I/O statements can refer to files that exist for the program at that point during execution. Some of the I/O statements (`INQUIRE`, `OPEN`, `CLOSE`, `WRITE`, `PRINT`, `REWIND`, and `ENDFILE`) can refer to files that do not exist. A `WRITE` or `PRINT` statement can create a file that does not exist and put data into that file, unless an error condition occurs.

An internal file always exists.

File position
9.1.3

Each file being processed by a program has a *position.* During the course of program execution, records are read or written, causing the file position to change. Also, there are other Fortran statements that cause the file position to change; an example is the `BACKSPACE` statement. The action produced by the I/O statements is described in terms of the file position, so it is important that file position be discussed in detail.

The *initial point* is the point just before the first record. The *terminal point* is the point just after the last record. If the file is empty, the initial point and the terminal point are the same. A file position can become indeterminate, in particular, when an error condition occurs. When the file position becomes indeterminate, the programmer cannot rely on the file being in any particular position.

A file can be positioned between records. In the example pictured in Figure 22, the file is positioned between records 2 and 3. In this case, record 2 is the *preceding record* and record 3 is the *next record.* Of course, if a file is positioned at its initial point, there is no preceding record, and there is no next record if it is positioned at its terminal point.

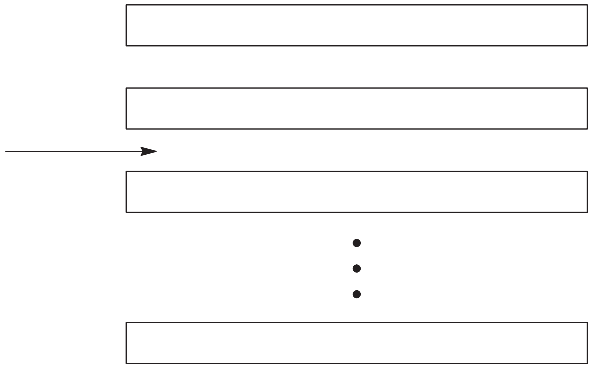

Figure 22. A file positioned between records

There may be a current record during execution of an I/O statement or after completion of a nonadvancing I/O statement as shown in Figure 23, where record 2 is the current record. If the file is positioned within a current record, the preceding record is the record immediately previous to the current record, unless the current record is also the initial record, in which case there is no preceding record. Similarly, the next record is the record immediately following the current record, unless the current record is also the final record in which case there is no next record.

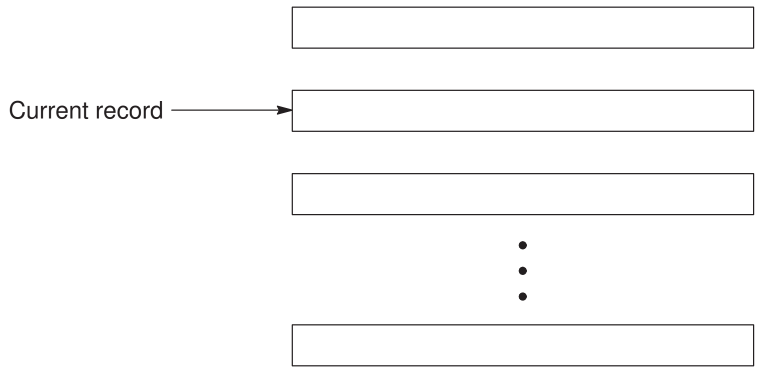

Figure 23. A file positioned with a current record

When there is a current record, the file is positioned at the initial point of the record, between values in the record, or at the terminal point of the record.

An internal file is always positioned at the beginning of a record just prior to data transfer.

Advancing I/O is record oriented; completion of such an operation always positions a file at the end of a record or between records, unless an error condition occurs. In contrast, nonadvancing I/O is character oriented; after reading and writing, the file can be positioned between characters within the current record.

While reading a file, the position of a nonadvancing file depends on whether success, data transfer, error, end-of-file, or end-of-record conditions occur while reading the file. The file position is indeterminate following an error condition when reading a file.

While writing a file, the position of a nonadvancing file depends on whether success, data transfer, or error conditions occur while writing the file. The file position is indeterminate following an error condition when writing a file.

When a nonadvancing input operation is performed, the file can be positioned after the last character of the file and before the logical or physical end-of-record. A subsequent nonadvancing input operation causes an end-of-record condition to occur, regardless of whether this record is the last record of the file. If another read operation is executed after the end-of-record condition occurs and the record is the last record of the file, an end-of-file condition occurs.

File access methods
9.1.4

There are two access file methods:

- Sequential access
- Direct access

Some files can be accessed by both methods; other files can be restricted to one access method or the other. For example, a magnetic tape can be accessed only sequentially. While each file is connected, it has a set of permissible access methods, which usually means that it can be accessed either sequentially or directly. However, a file must not be connected for both direct and sequential access simultaneously; that is, if a file is connected for direct access, it must be disconnected with a `CLOSE` statement and reconnected with an `OPEN` statement specifying sequential access before it can be referenced in a sequential access data transfer statement, and vice versa.

The actual file access method used to read or write the file is not a property of the file itself, but is indicated when the file is connected to a unit or when the file is created, if the file is preconnected. The same file can be accessed sequentially by a program, then disconnected, and then later accessed directly by the same program, if both types of access are permitted for the file.

Sequential access
9.1.4.1

Sequential access to the records in the file begins with the first record of the file and proceeds sequentially to the second record, and then to the next record, record-by-record. The records are accessed serially as they appear in the file. It is not possible to begin at some particular record within the file without reading down to that record in sequential order.

When a file is being accessed sequentially, the records are read and written sequentially . For example, if the records are written in any arbitrary order using direct access (see below) and then read using sequential access, the records are read beginning with record number 1 of the file, regardless of when it was written.

9.1.4.2 Direct access

When a file is accessed directly, the records are selected by record number. Using this identification, the records can be read or written in any order. Therefore, it is possible to write record number 47 first, then number 13. Reading or writing such records is accomplished by direct access data transfer I/O statements. Either record can be written without first accessing the other.

A file can be accessed by using both the direct and sequential access methods (but not both at the same time). However, direct access reads are restricted to records that have been written, and direct access writes are restricted to files connected for direct access. If a file contains an end-of-file record and is connected for direct access, the end-of-file record is not considered part of the file. If the sequential access method is not an allowed access method between the unit and the file, the file must not contain an end-of-file record.

A file can be created to support sequential or direct access, using the `ACCESS` specifier on the `OPEN` statement. A file can support both kinds of access if it is closed and reopened with a different specification and if records are of equal length. If a file supports both methods of access, the first record accessed sequentially is the record numbered 1 for direct access; the second sequential record is record 2 for direct access, and so on.

While a file is connected for one kind of access (sequential or direct), only I/O statements using that kind of access can be used with the file.

9.1.5 Units

I/O statements refer to a particular file by providing an *I/O unit*. An I/O unit is either an external unit or an internal unit. An external unit is either a nonnegative integer or an asterisk (`*`). When an external file is a nonnegative integer, it is called an *external file unit.* Table 23, page 372, lists the CF90 unit numbers.

A unit number identifies one and only one external unit in all program units in a Fortran program at a time.

File positioning, file connection, and inquiry statements must use an external unit.

If you are running the CF90 compiler on a Cray PVP system or a CRAY T3E system, an *external_unit_name* can be used to designate an external file unit; if you use an external name, it must be from 1 to 7 characters in length and be enclosed in apostrophes or quotation marks. On SPARC systems, the literal is limited to 3 characters.

ANSI/ISO: The Fortran 90 standard does not describe the use of external names to designate a unit.

An internal unit must be a default character variable. The name of an internal file also is called a unit.

Unit existence 9.1.5.1

The collection of unit numbers that can be used in a program for external files consists of the integers from 0 through 299. The unit numbers that can be used are said to *exist.* I/O statements must refer to units that exist, except for those that close a file or inquire about a unit. The assignments for unit numbers 100, 101, and 102 cannot be changed, and these numbers cannot appear in an `OPEN` statement. Numbers 0, 5, and 6 are not equivalent to the asterisk, and they can be reassigned.

The following table shows the unit numbers available to CF90 compiler users:

Table 23. CF90 unit numbers

Unit number	Availability
0	Associated with the `stderr` file or available for user specification
1–4	Available for user specification

Table 23. CF90 unit numbers (continued)

Unit number	Availability
5	Associated with the `stdin` file or available for user specification
6	Associated with `stdout` or available for user specification
7–99	Available for user specification
100	Reserved for system use as the `stdin` file
101	Reserved for system use as the `stdout` file
102	Reserved for system use as the `stderr` file
103–104	Reserved for system use
105–299	Available for user specification

Establishing a connection to a unit
9.1.5.2

In order to transfer data to or from an external file, the file must be connected to a unit. An internal file is always connected to the unit that is the name of the character variable. There are two ways to establish a connection between a unit and an external file:

- Execution of an `OPEN` statement in the executing program
- Preconnection by the operating system

Only one file can be connected to a unit at any given time and vice versa. If the unit is disconnected after its first use on a file, it can be reconnected later to another file or to the same file. A file that is not connected to a unit must not be used in any statement, except the `OPEN`, `CLOSE`, or `INQUIRE` statements.

Some units can be *preconnected* to files for each Fortran program by the operating system without any action necessary by the program. For example, units 5 and 6 are preconnected to files `stdin` and `stdout`, respectively . You also can use the `assign`(1) command to preconnect units. In either of these cases, the user program does not require an `OPEN` statement to connect a file for sequential access; it is preconnected.

Once a file has been disconnected, the only way to reference it is by its name using an `OPEN` or `INQUIRE` statement. There is no means of referencing an unnamed file once it is disconnected.

Data transfer statements

9.2

When a unit is connected, either by preconnection or execution of an `OPEN` statement, data can be transferred by reading and writing to the file associated with the unit. The transfer can occur to or from internal or external files.

The *data transfer statements* are the `READ`, `WRITE`, and `PRINT` statements. This subsection presents the general form of the data transfer statements first, followed by the forms that specify the major uses of data transfer statements.

The `READ`, `WRITE`, and `PRINT` statements are defined as follows:

R909	*read_stmt*	**is**	`READ (` *io_control_spec_list* `)` [*input_item_list*]
			`READ` *format* [`,` *input_item_list*]
R910	*write_stmt*	**is**	`WRITE (` *io_control_spec_list* `)` [*output_item_list*]
CRI		**or**	`WRITE` *format* [`,` *output_item_list*]
R911	*print_stmt*	**is**	`PRINT` *format* [`,` *output_item_list*]

ANSI/ISO: The Fortran 90 standard does not specify the `WRITE` *format* [`,` *output_item_list*] form of the `WRITE` statement.

The *io_control_spec_list* , the *input_item_list* , the *output_item_list* , and the *format* variables are described in subsequent subsections. The format item is called a *format specifier.*

The I/O control specifiers
9.2.1

The I/O control specifiers are defined as follows:

R912	*io_control_spec*	**is**	[`UNIT` =] *io_unit*
		or	[`FMT` =] *format*
		or	[`NML` =] *namelist_group_name*
		or	`REC` = *scalar_int_expr*
		or	`IOSTAT` = *scalar_default_int_variable*
		or	`ERR` = *label*
		or	`END` = *label*
		or	`ADVANCE` = *scalar_char_expr*
		or	`SIZE` = *scalar_default_int_variable*
		or	`EOR` = *label*

The `UNIT=` specifier, with or without the keyword `UNIT`, is called a *unit specifier*.

The `FMT=` specifier, with or without the keyword `FMT`, is called a *format specifier*.

The `NML=` specifier, with or without the keyword `NML`, is called a *namelist specifier*.

The data transfer statement is called a *formatted I/O statement* if a format or namelist group name specifier is present; it is called an *unformatted I/O statement* if neither is present. If a namelist group name specifier is present, it is called a *namelist I/O statement.* It is called a *direct access I/O statement* if a `REC=` specifier is present; otherwise, it is called a *sequential access I/O statement.*

The I/O control specification list must contain a unit specifier and can contain any of the other I/O control specifiers (but none can appear more than once). Both an `FMT=` and an `NML=` specifier can not appear in the same list.

This section describes the form and effect of the control information specifiers that are used in the data transfer statements. The `NML=`, `ADVANCE=`, `END=`, `EOR=`, `REC=`, and `SIZE=` specifiers are each unique to one of the forms of the data transfer statements, whereas the other specifiers

Section 9

are used in more than one form. In particular, NML= is used in the namelist data transfer statement; the ADVANCE=, EOR=, and SIZE= specifiers are used in input data transfer statements to specify nonadvancing formatted sequential data transfer; and the REC= specifier is used for direct access data transfer.

9.2.1.1 UNIT= control specifier

The format of the UNIT= control specifier is as follows:

```
[ UNIT = ] io_unit
```

An *io_unit* is defined as follows:

R901	*io_unit*	**is**	*external_file_unit*
		or	*
		or	*internal_file_unit*
CRI		**or**	*unit_name*
R902	*external_file_unit*	**is**	*scalar_int_expr*
R903	*internal_file_unit*	**is**	*char_variable*

ANSI/ISO: The Fortran 90 standard does not specify the use of a *unit_name* .

If you specify a *scalar_int_expr* for *io_unit* , it must be nonnegative and in the range of 0 through 299. Units 100 through 104 are reserved for the system.

Specifying an asterisk (*) for *io_unit* indicates a default external unit. It is the same unit number that would be defined if a READ or PRINT statement appeared without the unit number. The *io_unit* specified by an asterisk can be used only for formatted sequential access. The external unit used for a READ statement without a unit specifier or a READ statement with an asterisk unit specifier need not be the same as that used for a PRINT statement or WRITE statement with an asterisk unit specifier.

If you are using the CF90 compiler on a Cray PVP system or a CRAY T3E system, specifying an *external_unit_name* indicates a file name. The name can consist of 1 to 7 left-justified, blank-filled, or zero-filled ASCII characters. The use of uppercase and lowercase is significant for file names. The name is restricted to 3 characters on SPARC systems.

A unit specifier is required. A unit number identifies one and only one external unit in all program units in a Fortran program.

If the `UNIT` keyword is omitted, the I/O unit must be first. In this case, the keyword `FMT` or `NML` can be omitted from the format or namelist specifier. If the format or namelist specifier appears in the list, it must be the second specifier in the list.

`FMT=` control specifier 9.2.1.2

The format of the `FMT=` control specifier is as follows:

[`FMT` =] *format*

format is defined as follows:

R913	*format*	**is**	*char_expr*
		or	*label*
		or	*
		or	*scalar_default_int_variable*

Specifying a *char_expr* , which provides the format specification in the form of a character string, indicates formatted I/O. The character expression must be a valid format specification. If the expression is an array, it is treated as if all elements of the array were concatenated together in array element order and must be a valid format specification.

Specifying a *label* , which provides the statement label of a `FORMAT` statement containing the format specification, indicates formatted I/O. The *label* must be the label of a `FORMAT` statement in the same scoping unit as the data transfer statement.

Specifying a *scalar_default_int_variable* , which provides an integer variable that has been assigned the label of a FORMAT statement, indicates formatted I/O. The label value must have been assigned to the integer variable using an ASSIGN statement and must be the label of a FORMAT statement in the same scoping unit as the data transfer statement. For information on the FORMAT statement, see subsection 10.1.1, page 459.

Specifying an asterisk (*) indicates list-directed formatting.

The keyword FMT= can be omitted if the format specifier is the second specifier in the control information list and if the UNIT keyword was also omitted; otherwise, it is required.

If a format specifier is present, a namelist specifier (NML=) must not be present.

NML= control specifier 9.2.1.3

The format of the NML= control specifier is as follows:

```
[ NML = ] namelist_group_name
```

For *namelist_group_name* , specify the name of a namelist group declared in a NAMELIST statement. The namelist group name identifies the list of data objects to be transferred by the READ or WRITE statement with the NML= specifier .

If a namelist specifier is present, a format specifier must not be present.

ADVANCE= control specifier 9.2.1.4

The format of the ADVANCE= control specifier is as follows:

```
ADVANCE = scalar_char_expr
```

For *scalar_char_expr* , specify YES or NO. NO indicates nonadvancing formatted sequential data transfer. YES indicates advancing formatted sequential data transfer.

Section 9

Trailing blanks in the character expression are ignored. The value of the specifier is without regard to case (upper or lower); that is, the value `no` is the same as `NO`.

If an `ADVANCE=` specifier appears in the control information list, the data transfer must be a formatted sequential data transfer statement connected to an external unit. List-directed and namelist I/O are not allowed.

If the `EOR=` or `SIZE=` specifier appears in the control information list, an `ADVANCE=` specifier must also appear with the value `NO`.

`END=` control specifier 9.2.1.5

The format of the `END=` control specifier is as follows:

```
END = label
```

If an end-of-file condition occurs and no error condition occurs during the execution of the `READ` statement, the program branches to the label in the `END=` specifier. The label must be the label of a branch target statement in the same scoping unit as the `READ` statement.

The `END=` specifier can appear only in a sequential access `READ` statement; note that the `END=` specifier must not appear in a `WRITE` statement.

If the file is an external file, it is positioned after the end-of-file record.

If an `IOSTAT=` specifier is present and end-of-file condition occurs, but no error condition occurs, the `IOSTAT` variable specified becomes defined with a negative value.

`EOR=` control specifier 9.2.1.6

The format of the `EOR=` control specifier is as follows:

```
EOR = label
```

The program branches to the labeled statement specified by the `EOR=` specifier if an end of record is encountered for a nonadvancing `READ` statement. The label must be the label of a branch target statement in the same scoping unit as the `READ` statement.

The `EOR=` specifier can appear only in a `READ` statement with an `ADVANCE=` specifier with a value of `NO`, that is, a nonadvancing `READ` statement.

If an end-of-record condition occurs and no error condition occurs during the execution of the `READ` statement:

- The file is positioned after the current record.
- The variable given in the `IOSTAT=` specifier, if present, becomes defined with a negative value.
- If the connection has been made with the `PAD=` specifier of `YES`, the record is padded with blanks to satisfy both the input item list and the corresponding data edit descriptor if the input item and/or data edit descriptor requires more characters than are provided in the record.
- The variable given in the `SIZE=` specifier, if present, becomes defined with an integer value equal to the number of characters read from the input record; however, blank padding characters inserted because the `PAD=` specifier is `YES` are not counted.
- Execution of the `READ` statement terminates, and the program branches to the label in the `EOR=` specifier.

`ERR=` control specifier
9.2.1.7

The format of the `ERR=` control specifier is as follows:

```
ERR = label
```

If an error condition occurs, the position of the file becomes indeterminate.

The program branches to the label in the `ERR=` specifier if an error occurs in a data transfer statement. The label must be the label of a branch target statement in the same scoping unit as the data transfer statement.

If an `IOSTAT=` specifier is also present and an error condition occurs, the `IOSTAT` variable specified becomes defined with a positive value.

If the data transfer statement is a `READ` statement, contains a `SIZE=` specifier , and an error condition occurs, then the variable specified by the `SIZE=` specifier becomes defined with an integer value equal to the number of characters read from the input record; however, blank padding characters inserted because the `PAD=` specifier is `YES` are not counted.

`IOSTAT=` control specifier 9.2.1.8

The format of the `IOSTAT=` control specifier is as follows:

`IOSTAT =` *scalar_default_int_variable*

An integer value is returned in *scalar_default_int_variable* . The meaning of the return value is as follows:

- If *scalar_default_int_variable* > `0`, an error condition occurred.
- If *scalar_default_int_variable* = `0`, no error, end-of-file, or end-of-record condition occurred.
- If *scalar_default_int_variable* < 0, an end-of-file or end-of-record condition occurred. The end-of-file negative value is not the same as the negative value indicating the end-of-record condition.

The `IOSTAT=` specifier applies to the execution of the data transfer statement itself.

The variable specified in the `IOSTAT=` specifier must not be the same as or associated with any entity in the I/O item list or in the namelist group or with the variable specified in the `SIZE=` specifier, if present.

If the variable specified in the `IOSTAT=` specifier is an array element, its subscript values must not be affected by the data transfer, by any implied- `DO` item or processing, or with the definition or evaluation of any other specifier in the control specifier list.

You can obtain an online explanation of an error identified by the `IOSTAT` variable value. To do this, join the returned value with its group name, shown in the following list, and use the resulting string as an argument to the `explain`(1) command. For example:

```
explain lib-1250
explain 90476
```

Table 24. Message number identifiers

Message number	Group name	Source of message
1 through 899	`sys`	Operating system
1000 through 1999	`lib`	Fortran I/O library
5000 through 5999	`lib`	Flexible File I/O (FFIO) library
90000 through 90500	None	Tape system

9.2.1.9 `REC=` control specifier

The format of the `REC=` control specifier is as follows:

```
REC = scalar_int_expr
```

scalar_int_expr specifies an integer value that indicates the record number to be read or written.

The `REC=` specifier can appear only in a data transfer statement with a unit that is connected for direct access.

If the `REC=` specifier is present in a control information list, an `END=` or format specifier with an asterisk (for list-directed data transfer) must not be specified in the same control information list.

`SIZE=` control specifier 9.2.1.10

The format of the `SIZE=` control specifier is as follows:

`SIZE` = *scalar_default_int_variable*

The I/O system returns a nonnegative integer value in *scalar_default_int_variable* that indicates the number of characters read.

The `SIZE=` specifier applies to the execution of the `READ` statement itself and can appear only in a `READ` statement with an `ADVANCE=` specifier with the value `NO`.

Blanks inserted as padding characters when the `PAD=` specifier is `YES` for the connection are not counted.

The variable specified in the `SIZE=` specifier must not be the same as or associated with any entity in the I/O item list or in the namelist group or with the variable specified in the `IOSTAT=` specifier, if present.

If the variable specified in the `SIZE=` specifier is an array element, its subscript values must not be affected by the data transfer, by any implied- `DO` item or processing, or with the definition or evaluation of any other specifier in the control specifier list.

The I/O item list 9.2.2

The I/O item list consists basically of a list of variables in a `READ` statement and a list of expressions in a `WRITE` or `PRINT` statement. In addition, in any of these statements, the I/O item list can contain an I/O implied-`DO` list, containing a list of variables or expressions indexed by `DO` variables.

The components of the I/O lists are defined as follows:

R914	*input_item*	**is**	*variable*
		or	*io_implied_do*
R915	*output_item*	**is**	*expr*
		or	*io_implied_do*
R916	*io_implied_do*	**is**	(*io_implied_do_object_list* , ≡
			≡ *io_implied_do_control*)
R917	*io_implied_do_object*	**is**	*input_item*
		or	*output_item*
R918	*io_implied_do_control*	**is**	*do_variable* = *scalar_numeric_expr* , ≡
			≡ *scalar_numeric_expr* [, *scalar_numeric_expr*]

The `DO` variable must be a scalar integer or real variable. If it is real, it must be default real or double-precision real; the use of real `DO` variables is considered obsolescent.

Each scalar numeric expression must be of type integer or real. If it is of type real, each must be of type default real or default double precision; the use of such real expressions is considered obsolescent. They need not be all of the same type nor of the type of the `DO` variable.

The `DO` variable must not be one of the input items in the implied-`DO`; it must not be associated with an input item either.

Two nested implied-`DO`s must not have the same (or associated) `DO` variables.

An implied-`DO` object, when it is part of an input item, must itself be an input item; that is, it must be a variable or an implied-`DO` object whose objects are ultimately variables. Similarly, an implied-`DO` object, when it is part of an output item, must itself be an output item; that is, it must be an expression or an implied-`DO` object whose objects are ultimately expressions.

For an I/O implied-DO, the loop is initialized, executed, and terminated in the same manner as for the DO construct. Its iteration count is established at the beginning of processing of the items that constitute the I/O implied- DO.

An array appearing without subscripts in an I/O list is treated the same as if all elements of the array appeared in array-element order. In the following example, assume UP is an array of shape (2,3):

```
READ *, UP
```

The preceding statement is the same as the following code:

```
READ *, UP(1,1), UP(2,1), UP(1,2),  &
        UP(2,2), UP(1,3), UP(2,3)
```

When a subscripted array is an input item, it is possible that when a value is transferred from the file to the variable, it might affect another part of the input item. This is not allowed. Consider the following READ statements, for example:

```
INTEGER  A(100), V(10)

READ *, A(A)
READ *, A(A(1):A(9))
! Suppose V's elements are defined with
! values in the range 1 to 100.
READ *, A(V)
```

The first two READ statements are not valid because the data values read affect other parts of array A. The third READ statement is allowed as long as no two elements of V have the same value.

Assumed-size arrays cannot appear in I/O lists, unless a subscript, a section subscript specifying an upper bound, or a vector subscript appears in the last dimension.

In formatted I/O, a structure is treated as if, in place of the structure, all components were listed in the order of the components in the derived-type definition. In the following statement, assume that FIRECHIEF is a structure of type PERSON (as defined in subsection 4.6, page 125):

```
READ *, FIRECHIEF
```

The preceding statement is the same as the following:

```
READ *, FIRECHIEF%AGE, FIRECHIEF%NAME
```

A pointer can be an I/O list item but it must be associated with a target at the time the data transfer statement is executed. For an input item, the data in the file is transferred to the associated target. For an output item, the target associated with the pointer must be defined, and the value of the target is transferred to the file.

The following types of items cannot be used in an I/O list: a structure with an ultimate component that is a pointer, auxiliary variables, or auxiliary arrays.

> **ANSI/ISO:** The Fortran 90 standard does not specify auxiliary variables or auxiliary arrays.

A constant or an expression with operators, parentheses, or function references cannot appear as an input list item, but can appear as an output list item. A function reference in an output list must not cause execution of another I/O statement on this same unit.

> **ANSI/ISO:** The Fortran 90 standard does not allow the execution of another I/O statement on any unit through a function reference in an I/O list. For more information on restrictions, see subsection 9.9, page 453.

An input list item, or an entity associated with it, must not contain any portion of an established format specification.

On output, every entity whose value is to be written must be defined.

Explicitly formatted advancing sequential access data transfer
9.2.3

For formatted input and output, the file consists of characters. These characters are converted into representations suitable for storing in the computer memory during input and converted from an internal representation to characters on output. When a file is accessed sequentially, records are processed in the order in which they appear in the file.

The explicitly formatted, advancing, sequential access data transfer statements have the following formats:

```
READ ( [ UNIT = ] io_unit , &
[ FMT = ] format &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] &
[ , END = label ] &
[ , ADVANCE = 'YES' ] ) &
[ input-item-list ]
```

```
READ format [ , input_item_list ]
```

```
WRITE ( [ UNIT = ] io_unit , &
[ FMT = ] format &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] &
[ , ADVANCE = 'YES' ] ) &
[ output_item_list ]
```

```
PRINT format [ , output_item_list ]
```

```
WRITE format [ , output_item_list ]
```

Section 9

The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (*), or a character literal constant (external name). All three I/O unit forms indicate that the unit is a formatted sequential access external unit.

> **ANSI/ISO:** The Fortran 90 standard does not specify the use of a character literal constant (external name) as an I/O unit.

If an `ADVANCE=` specifier is present, the format must not be an asterisk (*).

When an advancing I/O statement is executed, reading or writing of data begins with the next character in the file. If a previous I/O statement was a nonadvancing statement, the next character transferred may be in the middle of a record, even if the statement being executed is an advancing statement. The essential difference between advancing and nonadvancing sequential data transfer is that an advancing I/O statement always leaves the file positioned at the end of the record.

During the data transfer, data is transferred with editing between the file and the entities specified by the I/O list. Format control is initiated and editing is performed as described in section 10. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For input data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

For input data transfer, the input record is logically padded with blanks to satisfy an input list and format specification that requires more characters than the record contains, unless the `PAD=` specifier is specified as `NO` in the `OPEN` statement. If the `PAD=` specifier is `NO`, the input list and format specification must not require more characters from the record than the record contains.

For output data transfer, the output list and format specification must not specify more characters for a record than the record size; recall that the record size for an external file is specified by a `RECL=` specifier in the `OPEN` statement.

If the file is connected for formatted I/O, unformatted data transfer is prohibited.

Execution of an advancing sequential access data transfer statement terminates when one of the following occurs:

- Format processing encounters a data or colon edit descriptor, and there are no remaining elements in the input item list or output item list
- On input, an end-of-file condition is encountered
- An error condition is encountered

Example of formatted `READ` statements are as follows:

```
! Assume that FMT_5 is a character string
! whose value is a valid format specification.
READ (5, 100, ERR=99, END=200)  &
               A, B, (C(I), I = 1, 40)
READ (9, IOSTAT=IEND, FMT=FMT_5) X, Y
READ (FMT="(5E20.0)", UNIT=5,  &
      ADVANCE="YES") (Y(I), I = 1, KK)
READ 100, X, Y
```

Examples of `WRITE` statements are as follows:

```
! Assume FMT_103 is a character string with a
! valid format specification.
WRITE (9, FMT_103, IOSTAT=IS, ERR=99) A, B, C, S
WRITE (FMT=105, ERR=9, UNIT=7)  X
WRITE (*, "(F10.5)")  X
PRINT "(A, E14.6)", " Y = ", Y
```


Unformatted sequential access
9.2.4

For unformatted sequential input and output, the file consists of values stored using an internal binary representation. This means that little or no conversion is required during input and output.

Unformatted sequential access data transfer statements are the READ and WRITE statements with no format specifier or namelist group name specifier . The formats are as follows:

```
READ ( [ UNIT = ] scalar_int_expr  &
   [ , IOSTAT = scalar_default_int_variable  ] &
   [ , ERR = label ] &
   [ , END = label ] ) &
   [ input_item_list  ]
```

```
WRITE ( [ UNIT = ] scalar_int_expr  &
   [ , IOSTAT = scalar_default_int_variable  ] &
   [ , ERR = label ] ) &
   [ output_item_list  ]
```

Data is transferred without editing between the current record and the entities specified by the I/O list. Exactly one record is read or written.

Objects of intrinsic or derived types can be transferred through an unformatted data transfer statement.

For input data transfer, the file must be positioned so that the record read is an unformatted record or an end-of-file record.

For input data transfer, the number of values required by the input list must be less than or equal to the number of values in the record. Each value in the record must be of the same type as the corresponding entity in the input list, except that one complex value may correspond to two real list entities or two real values may correspond to one complex list entity. The type parameters of the corresponding entities must be the same; if two real values correspond to one complex entity or one complex value corresponds to two real entities, all three must have the same kind type parameter values. If an entity in the input list is of type character, the character entity must have the same length as the character value.

On output, if the file is connected for unformatted sequential access data transfer, the record is created with a length sufficient to hold the values from the output list. This length must be one of the set of allowed record lengths for the file and must not exceed the value specified in the RECL= specifier, if any, of the OPEN statement that established the connection.

Execution of an unformatted sequential access data transfer statement terminates when one of the following occurs:

- The input item list or output item list is exhausted
- On input, an end-of-file condition is encountered
- An error condition is encountered

If the file is connected for unformatted I/O, formatted data transfer is prohibited.

The following are examples of unformatted sequential access READ statements:

```
READ (5, ERR=99, END=100)  A, B, (C(I), I = 1, 40)
READ (IOSTAT=IEND, UNIT=9) X, Y
READ (5) Y
```

The following are examples of unformatted sequential access WRITE statements:

```
WRITE (9, IOSTAT=IS, ERR=99) A, B, C, S
WRITE (ERR=99, UNIT=7) X
WRITE (9) X
```

If the access is sequential, the file is positioned at the beginning of the next record prior to data transfer and positioned at the end of the record when the I/O is finished, because nonadvancing unformatted I/O is not permitted.

Nonadvancing formatted sequential data transfer
9.2.5

Nonadvancing formatted sequential I/O provides the capability of reading or writing part of a record. It leaves the file positioned after the last character read or written, rather than skipping to the end of the record. Processing of nonadvancing input continues within a current record, until an end-of-record condition occurs. Nonadvancing input statements can read varying-length records and determine their lengths. Nonadvancing I/O is sometimes called *partial record* or *stream I/O*. It may be used only with explicitly formatted, external files connected for sequential access.

The formats for the nonadvancing I/O statements are as follows:

```
READ ( [ UNIT = ] io_unit , &
[ FMT = ] format , &
ADVANCE = 'NO' &
[ , SIZE = scalar_default_int_variable ] &
[ , EOR = label ] &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] &
[ , END = label ] ) &
[ input_item_list ]
```

```
WRITE ( [ UNIT = ] io_unit , &
[ FMT = ] format , &
ADVANCE = 'NO' &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] ) &
[ output_item_list ]
```


The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (*), or a character literal constant (external name). The I/O unit forms indicate that the unit is a formatted sequential access external unit.

The format must not be an asterisk (*).

During the data transfer, data is transferred with editing between the file and the entities specified by the I/O list. Format control is initiated and editing is performed as described in section 10. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For input data transfer, the file must be positioned at the beginning of, end of, or within a formatted record or at the beginning of an end-of-file record.

For input data transfer, the input record is logically padded with blanks to satisfy an input list and format specification that requires more characters than the record contains, unless the `PAD=` specifier is specified as `NO` in the `OPEN` statement. If the `PAD=` specifier is `NO`, the input list and format specification must not require more characters from the record than the record contains, unless either the `EOR=` or `IOSTAT=` specifier is present. In the exceptional cases during nonadvancing input, the actions and execution sequence are described in subsection 9.4, page 414.

For output data transfer, the output list and format specification must not specify more characters for a record than the record size; recall that the record size for an external file may be specified by a `RECL=` specifier in the `OPEN` statement.

The variable in the `SIZE=` specifier is assigned the number of characters read on input. Blanks inserted as padding characters when the `PAD=` specifier is `YES` are not counted.

The program branches to the label given by the `EOR=` specifier, if an end-of-record condition is encountered during input. The label must be the label of a branch target statement in the same scoping unit as the data transfer statement.

Execution of a nonadvancing, formatted, sequential access data transfer statement terminates when one of the following occurs:

- Format processing encounters a data edit descriptor or colon edit descriptor , and there are no remaining elements in the input item list or output item list
- On input, an end-of-file or end-of-record condition is encountered
- An error condition is encountered

Unformatted data transfer is prohibited.

Section 9

In the following statements, assume that N has the value 7:

```
WRITE (*, '(A)', ADVANCE="NO") "The answer is "
PRINT '(I1)', N
```

The preceding statements produce the following single output record:

```
The answer is 7
```

In the following statements, assume that SSN is a rank-one array of size 9 with values (1, 2, 3, 0, 0, 9, 8, 8, 6):

```
DO I = 1, 3
  WRITE (*, '(I1)', ADVANCE="NO") SSN(I)
ENDDO
WRITE (*, '("-")', ADVANCE="NO")
DO I = 4, 5
  WRITE (*, '(I1)', ADVANCE="NO") SSN(I)
ENDDO
WRITE (*, '(A1)', ADVANCE="NO")  '-'
DO I = 6, 9
  WRITE (*, '(I1)', ADVANCE="NO") SSN(I)
ENDDO
```

The preceding statements produce the following record:

```
123-00-9886
```


Direct access data transfer

9.2.6

In direct access data transfer, the records are selected by record number. The record number is a scalar integer expression whose value represents the record number to be read or written. The records can be written in any order , but all records must be of the length specified by the RECL= specifier in an OPEN statement.

If a file is connected using the direct access method, then nonadvancing, list-directed, and namelist I/O are prohibited. Also, an internal file must not be accessed using the direct access method.

It is not possible to delete a record using direct access. However, records may be rewritten so that a record can be erased by writing blanks into it.

9.2.6.1 Formatted direct access data transfer

For formatted input and output, the file consists of characters. These characters are converted into internal representation during input and are converted from an internal representation to characters on output.

Formatted direct access data transfer statements are `READ` and `WRITE` statements with a `REC=` specifier and a format specifier. The formats for formatted direct access data transfer statements are as follows:

```
READ ( [ UNIT = ] scalar_int_expr , &
[ FMT = ] format , &
REC = scalar_int_expr &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] ) &
[ input_item_list ]
```

```
WRITE ( [ UNIT = ] scalar_int_expr , &
[ FMT = ] format , &
REC = scalar_int_expr &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] ) &
[ output_item_list ]
```

Section 9

The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (`*`), or a character literal constant (external name). The I/O unit forms indicate that the unit is a formatted sequential access external unit.

The format must not be an asterisk (`*`).

Note that the above forms are intentionally structured so that the unit cannot be an internal file, and the `END=`, `ADVANCE=`, and namelist specifiers cannot appear.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all entities specified by the input list to become undefined.

During the data transfer, data is transferred with editing between the file and the entities specified by the I/O list. Format control is initiated and editing is performed as described in section 10, which begins on page 455. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For input data transfer, the file must be positioned so that the record read is a formatted record.

For input data transfer, the input record is logically padded with blanks to satisfy an input list and format specification that requires more characters than the record contains, unless the `PAD=` specifier was specified as `NO` in the `OPEN` statement. If the `PAD=` specifier is `NO`, the input list and format specification must not require more characters from the record than the record contains.

For output data transfer, the output list and format specification must not specify more characters for a record than the record size; recall that the record size for an external file is specified by a `RECL=` specifier in the `OPEN` statement.

If the format specification specifies another record (say, by the use of the slash edit descriptor), the record number is increased by one as each succeeding record is read or written by that I/O statement.

For output data transfer, if the number of characters specified by the output list and format do not fill a record, blank characters are added to fill the record.

Execution of a formatted direct access data transfer statement terminates when one of the following events occurs:

- Format processing encounters a data edit descriptor or colon edit descriptor , and there are no remaining elements in the input item list or output item list
- An error condition is encountered

If the file is connected for formatted I/O, unformatted data transfer is prohibited.

Examples of formatted direct access I/O statements are as follows. In these examples, assume that `FMT_X` is a character entity.

```
READ (7, FMT_X, REC=32, ERR=99) A
READ (IOSTAT=IO_ERR, REC=34,  &
      FMT=185, UNIT=10, ERR=99)  A, B, D
WRITE (8, "(2F15.5)", REC = N + 2) X, Y
```

9.2.6.2 Unformatted direct access data transfer

For unformatted input and output, the file consists of values stored using a representation that is close to or the same as that used in program memory. This means that little or no conversion is required during input and output.

Unformatted direct access data transfer statements are `READ` and `WRITE` statements with a `REC=` specifier and no format specifier. The formats for unformatted direct access data transfer statements are as follows:

```
READ ( [ UNIT = ] scalar_int_expr , &
REC = scalar_int_expr  &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] ) &
[ input_item_list ]
```

Section 9

```
WRITE ( [ UNIT = ] scalar_int_expr , &
REC = scalar_int_expr &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] ) &
[ output_item_list ]
```

The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (`*`), or a character literal constant (external name). The I/O unit forms indicate that the unit is a formatted sequential access external unit.

Note that the preceding forms are intentionally structured so that the unit cannot be an internal file, and the `FMT=`, `END=`, `ADVANCE=`, and namelist specifiers cannot appear.

On input, an attempt to read a record of a file connected for direct access that has not previously been written causes all entities specified by the input list to become undefined.

The number of items in the input list must be less than or equal to the number of values in the input record.

Data is transferred without editing between the current record and the entities specified by the I/O list. Exactly one record is read or written.

Objects of intrinsic or derived types may be transferred.

For input data transfer, the file must be positioned so that the record read is an unformatted record.

For input data transfer, the number of values required by the input list must be less than or equal to the number of values in the record. Each value in the record must be of the same type as the corresponding entity in the input list, except that one complex value may correspond to two real list entities or two real values may correspond to one complex list entity. The type parameters of the corresponding entities must be the same; if two real values correspond to one complex entity or one complex value corresponds to two real entities, all three must have the same kind type parameter values. If an entity in the input list is of type character, the character entity must have the same length as the character value.

The output list must not specify more values than can fit into the record. If the file is connected for direct access and the values specified by the output list do not fill the record, the remainder of the record is undefined.

Execution of an unformatted direct access data transfer statement terminates when one of the following events occurs:

- The input item list or output item list is exhausted
- An error condition is encountered

If the file is connected for unformatted direct access I/O, formatted data transfer is prohibited.

The following are examples of unformatted direct access I/O statements:

```
READ (7, REC=32, ERR=99) A
READ (IOSTAT=MIS, REC=34, UNIT=10, ERR=99) A, B, D
WRITE (8, REC = N + 2) X, Y
```

List-directed data transfer

9.2.7

List-directed formatting can occur only with files connected for sequential access; however, the file can be an internal file. The I/O data transfer must be advancing. The records read and written are formatted.

List-directed data transfer statements are any data transfer statement for which the format specifier is an asterisk (*). The formats for the list-directed data transfer statements are as follows:

```
READ ( [ UNIT = ] io_unit , &
[ FMT = ] * &
[ , IOSTAT = scalar_default_int_variable ] &
[ , END = label ] &
[ , ERR = label ] ) &
[ input_item_list ]
```

Section 9

```
READ * [ , input-item-list ]
```

```
WRITE ( [ UNIT = ] io_unit , &
[ FMT = ] * &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] ) &
[ output_item_list ]
```

```
PRINT * [ , output_item_list ]
```

```
WRITE format [ , output_item_list ]
```

Note that the preceding forms are intentionally structured so that the `ADVANCE=` and namelist specifiers cannot appear.

During the data transfer, data is transferred with editing between the file and the entities specified by the I/O list. The rules for formatting the data transferred are discussed in subsection 10.11, page 499. The current record and, possibly, additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For input data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

For output data transfer, one or more records will be written. Additional records are written if the output list specifies more characters for a record than the record size.

If the file is connected for list-directed data transfer, unformatted data transfer is prohibited.

Execution of a list-directed data transfer statement terminates when one of the following conditions occurs:

- The input item list or the output item list is exhausted
- On input, an end-of-file is encountered, or a slash (/) is encountered as a value separator
- An error condition is encountered

The following are examples of list-directed input and output statements:

```
READ (5, *, ERR=99, END=100)  &
               A, B, (C(I), I = 1, 40)
READ (FMT=*, UNIT=5) (Y(I), I = 1, KK)
READ *, X, Y
WRITE (*, *) X
PRINT *, " Y = ", Y
```

Namelist data transfer

9.2.8

Namelist I/O uses a group name for a list of variables that are transferred. Before the group name can be used in the transfer, the list of variables must be declared in a NAMELIST statement, which is a specification statement. Using the namelist group name eliminates the need to specify the list of variables in the sequence for a namelist data transfer. The formatting of the input or output record is not specified in the program; it is determined by the contents of the record itself or the items in the namelist group. Conversion to and from characters is implicit for each variable in the list.

9.2.8.1 Form of a namelist group declaration

All namelist I/O data transfer statements use a namelist group name, which must be declared. As described in subsection 5.9, page 188, the format of the namelist group name declaration is as follows:

```
NAMELIST / namelist_group_name / variable_name &
  [ , variable_name ] ... [ [ , ] &
  / namelist_group_name / variable_name [ , variable_name ] ... ] ...
```

Examples:

```
NAMELIST /GOAL/ G, K, R
NAMELIST /XLIST/ A, B /YLIST/ Y, YY, YU
```

9.2.8.2 Forms of namelist I/O statements

Namelist input and output data transfer statements are `READ` and `WRITE` statements with a namelist specifier. The formats for namelist data transfer statements are as follows:

```
READ ( [ UNIT = ] io_unit , &
[ NML = ] namelist_group_name &
[ , IOSTAT = scalar_default_int_variable ] &
[ , END = label ] &
[ , ERR = label ] )
```

```
WRITE ( [ UNIT = ] io_unit , &
[ NML = ] namelist_group_name &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] )
```

Section 9

Note that the preceding forms do not contain the `REC=`, `FMT=`, and `ADVANCE=` specifiers. There must be no input or output item list.

The I/O unit must not be an internal file.

During namelist data transfer, data is transferred with editing between the file and the entities specified by the namelist group name. Format control is initiated and editing is performed as described in subsection 10.12, page 506. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

For namelist input data transfer, the file must be positioned so that the record read is a formatted record or an end-of-file record.

For namelist output data transfer, one or more records will be written. Additional records are written if the namelist specifies more characters for a record than the record size; recall that the record size for an external file can be specified by a `RECL=` specifier in the `OPEN` statement.

If an entity appears more than once within the input record for a namelist input data transfer, the last value is the one that is used.

For namelist input data transfer, all values following a *name* = part within the input record are transmitted before processing any subsequent entity within the namelist input record.

Execution of a namelist data transfer statement terminates when one of the following events occurs:

- On input, an end-of-file is encountered, or a slash (/) is encountered as a value separator
- On input, the end of the namelist input record is reached and a name-value subsequence has been processed for every item in the namelist group object list
- On output, the namelist group object list is exhausted
- An error condition is encountered

If the file is connected for namelist data transfer, unformatted data transfer is prohibited.

Examples of namelist data transfer statements are as follows:

```
READ (NML=NAME_LIST_23, IOSTAT=KN,  UNIT=5)
WRITE (6, NAME_LIST_23, ERR=99)
```

Data transfer on internal files
9.2.9

Transferring data from an internal representation to characters or from characters back to internal representation can be done between two variables in an executing program. A formatted sequential access input or output statement, including list-directed formatting, is used for the transfer. The format is used to interpret the characters. The internal file and the internal unit are the same character variable.

With this feature, it is possible to read in a string of characters without knowing its exact format, examine the string, and then interpret it according to its contents.

Formatted sequential access data transfer statements on an internal file have the following formats:

```
READ ( [ UNIT = ] char_variable , &
[ FMT = ] format &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] &
[ , END = label ] ) &
[ input_item_list ]
```

Section 9

```
WRITE ( [ UNIT = ] char_variable , &
[ FMT = ] format &
[ , IOSTAT = scalar_default_int_variable ] &
[ , ERR = label ] ) &
[ output_item_list ]
```

Examples of data transfer on internal files are:

```
READ (CHAR_124, 100, IOSTAT=ERR) MARY, X, J, NAME
WRITE (FMT=*, UNIT=CHAR_VAR) X
```

The unit must be a character variable that is not an array section with a vector subscript.

Each record of an internal file is a scalar character variable.

If the character variable is an array or an array section, each element of the array or section is a scalar character variable and thus a record. The order of the records is array element order. The length, which must be the same for each record, is the length of one array element.

If the character variable is an array or part (component, element, section, or substring) of an array that has the ALLOCATABLE attribute, the variable must be allocated before it is used as an internal file. It must be defined if it is used as an internal file in a READ statement.

If the character variable is a pointer, it must be associated with a target. The target must be defined if it is used as an internal file in a READ statement.

During data transfer, data is transferred with editing between the internal file and the entities specified by the I/O list. Format control is initiated and editing is performed as described in section 10, page 455. The current record and possibly additional records are read or written.

For the data transfer, values can be transmitted to or from objects of intrinsic or derived types. In the latter case, the transmission is in the form of values of intrinsic types to or from the components of intrinsic types, which ultimately comprise these structured objects.

Section 9

For output data transfer, the output list and format specification must not specify more characters for a record than the record size. Recall that the record size for an internal file is the length of the scalar character variable or an element in a character array that represents the internal file. The format specification must not be part of the internal file or associated with the internal file or part of it.

If the number of characters written is less than the length of the record, the remaining characters are set to blank.

The records in an internal file are defined when the record is written. An I/O list item must not be in the internal file or associated with the internal file. An internal file also can be defined by a character assignment statement, or some other means, or can be used in expressions in other statements. For example, an array element may be given a value with a `WRITE` statement and then used in an expression on the right-hand side of an assignment statement.

To read a record in an internal file, the character variable comprising the record must be defined.

Before a data transfer occurs, an internal file is positioned at the beginning of the first record (that is, before the first character, if a scalar, and before the first character of the first element, if an array). This record becomes the current record.

Only formatted sequential access, including list-directed formatting, is permitted on internal files. Namelist formatting is prohibited.

On input, an end-of-file condition occurs when there is an attempt to read beyond the last record of the internal file.

During input processing, all nonleading blanks in numeric fields are treated as if they were removed, right justifying all characters in the field (as if a `BN` edit descriptor were in effect). In addition, records are blank padded when an end of record is encountered before all of the input items are read (as if `PAD=YES` were in effect).

For list-directed output, character values are not delimited.

File connection, positioning, and inquiry must not be used with internal files.

Execution of a data transfer statement on an internal file terminates when one of the following events occurs:

- Format processing encounters a data or colon edit descriptor, and there are no remaining elements in the input item list or output item list
- If list-directed processing is specified, the input item list or the output item list is exhausted; or on input, a slash (/) is encountered as a value separator
- On input, an end-of-file condition is encountered
- An error condition is encountered

Printing formatted records

9.2.10

Sometimes output records are sent to a device that interprets the first character of the record as a control character. This is usually the case with line printers. If a formatted record is transferred to such a device, the first character of the record is not printed, but instead is used to control vertical spacing. The remaining characters of the record, if any, are printed on one line beginning at the left margin. This transfer of information is called *printing*.

The first character of such a record must be of character type and determines vertical spacing as specified in Table 25.

Table 25. Interpretation of the first character for printing control

Character	Vertical spacing before printing
Blank	One line (single spacing)
0	Two lines (double spacing)
1	To first line of next page (begin new page)
+	No advance (no spacing – printing on top of previous line)

Section 9

If there are no characters in the record, a blank line is printed. If the first character is not a blank, 0, 1, or +, the character is treated as a blank.

The `PRINT` statement does not imply that printing actually will occur on a printer, and the `WRITE` statement does not imply that printing will not occur. Whether printing occurs depends on the device connected to the unit number.

9.2.11 `BUFFER IN` and `BUFFER OUT` statements (CF90 extensions)

You can use the `BUFFER IN` and `BUFFER OUT` statements to transfer data.

> **ANSI/ISO:** The Fortran 90 standard does not describe the `BUFFER IN` or `BUFFER OUT` statements.

On Cray PVP systems and CRAY T3E systems, the data can be transferred while allowing the subsequent execution sequence to proceed concurrently; this is called asynchronous I/O. Asynchronous I/O requires the use of pure data (unblocked) files, as discussed in the *Application Programmer's I/O Guide*, publication SG–2168. `BUFFER IN` and `BUFFER OUT` operations can proceed simultaneously on several units or files. On CRAY T3E systems, asynchronous I/O is allowed when the operating system permits.

> **Note:** Asynchronous I/O is not available on SPARC systems. The I/O is mapped to synchronous I/O.

`BUFFER IN` is for reading, and `BUFFER OUT` is for writing. A `BUFFER IN` or `BUFFER OUT` operation includes only data from a single array or a single common block.

Either statement initiates a data transfer between a specified file or unit (at the current record) and memory. If the unit or file is completing an operation initiated by any earlier `BUFFER IN` or `BUFFER OUT` statement, the current `BUFFER IN` or `BUFFER OUT` statement suspends the execution sequence until the earlier operation is complete. When the unit's preceding operation terminates, execution of the `BUFFER IN` or `BUFFER OUT` statement completes as if no delay had occurred.

You can use the `UNIT` or `LENGTH` intrinsic procedures to delay the execution sequence until the `BUFFER IN` or `BUFFER OUT` operation is complete. These functions can also return information about the I/O operation at its termination. On Cray PVP systems and CRAY T3E systems, you can use the SETPOS library routine with BUFFER IN and BUFFER OUT for random positioning. For more information on the UNIT and LENGTH intrinsic procedures, see unit(7) or length(7). For more information on SETPOS, see the following manuals:

- *Application Programmer's Library Reference Manual*, publication SR–2165
- *Application Programmer's I/O Guide*, publication SG–2168
- *UNICOS System Libraries Reference Manual*, publication SR–2080

The BUFFER IN and BUFFER OUT statements are defined as follows.

The general format of the BUFFER IN and BUFFER OUT statements follows:

CRI	*buffer_in_stmt*	**is**	BUFFER IN (*id* , *mode*) (*start_loc* , *end_loc*)
CRI	*buffer_out_stmt*	**is**	BUFFER OUT (*id* , *mode*) (*start_loc* , *end_loc*)
CRI	*io_unit*	**is**	*external_file_unit*
		or	*file_name_expr*
CRI	*mode*	**is**	*scalar_integer_expr*
CRI	*start_loc*	**is**	*variable*
CRI	*end_loc*	**is**	*variable*

In the preceding definition, the *variable* specified for *start_loc* and *end_loc* cannot be of a derived type if you are performing implicit data conversion. The data items between *start_loc* and *end_loc* must be of the same type.

BUFFER IN (*io_unit* , *mode*) (*start_loc* , *end_loc*)

BUFFER OUT (*io_unit* , *mode*) (*start_loc* , *end_loc*)

Section 9

io_unit — An identifier that specifies a unit or a file name. The I/O unit is a scalar integer expression with a nonnegative value, an asterisk (`*`), or a character literal constant (external name). The I/O unit forms indicate that the unit is a formatted sequential access external unit.

mode — Mode identifier. This integer expression controls the record position following the data transfer. The mode identifier is ignored on files that do not contain records; only full record processing is available.

start_loc , *end_loc* — Symbolic names of the variables, arrays, or array elements that mark the beginning and ending locations of the `BUFFER IN` or `BUFFER OUT` operation. These names must be either elements of a single array (or equivalenced to an array) or members of the same common block. If *start_loc* or *end_loc* is of type character, then both must be of type character . If *start_loc* and *end_loc* are noncharacter, then the item length of each must be equal.

For example, if the internal length of the data type of *start_loc* is 64 bits, the internal length of the data type of *end_loc* must be 64 bits. To ensure that the size of *start_loc* and *end_loc* are the same, use the same data type for both.

The mode identifier, *m*, controls the position of the record at unit *io_unit* after the data transfer is complete. The values of *m* have the following effects:

- Specifying *mode* ≥ 0 causes full record processing. File and record positioning works as with conventional I/O. The record position following such a transfer is always between the current record (the record with which the transfer occurred) and the next record. Specifying `BUFFER OUT` with *mode* ≥ 0 ends a series of partial-record transfers.
- Specifying *mode* < 0 causes partial record processing. In `BUFFER IN`, the record is positioned to transfer its $(n+1)$th word if the nth word was the last transferred. In `BUFFER OUT`, the record is left positioned to receive additional words.

The amount of data to be transferred is specified in words without regard to types or formats. However, the data type of *end_loc* affects the exact ending location of a transfer. If *end_loc* is of a multiple-word data type, the location of the last word in its multiple-word form of representation marks the ending location of the data transfer.

`BUFFER OUT` with *start_loc* =*end_loc* + 1 and $m \geq 0$ causes a zero-word transfer and concludes the record being created. Except for terminating a partial record, *start_loc* following *end_loc* in a storage sequence causes a run-time error.

Example:

```
      PROGRAM XFR
      DIMENSION A(1000), B(2,10,100), C(500)
      ...
      BUFFER IN(32,0) (A(1),A(1000))
      ...
      DO 9 J=1,100
        B(1,1,J) = B(1,1,J) + B(2,1,J)
    9 CONTINUE
      BUFFER IN(32,0) (C(1),C(500))
      BUFFER OUT(22,0) (A(1),A(1000))
      ...
      END
```

The first `BUFFER IN` statement in this example initiates a transfer of 1000 words from unit 32. If asynchronous I/O is available, processing unrelated to that transfer proceeds. When this is complete, a second `BUFFER IN` is encountered, which causes a delay in the execution sequence until the last of the 1000 words is received. A transfer of another 500 words is initiated from unit 32 as the execution sequence continues. `BUFFER OUT` begins a transfer of the first 1000 words to unit 22. In all cases *mode* = 0, indicating full record processing.

Section 9

Execution model for data transfer statements

9.3

When a data transfer statement is executed, the following steps are taken:

1. Determine the direction of data transfer. A `READ` statement indicates that data is to be transferred from a file to program variables. A `WRITE` or `PRINT` statement indicates that data is to be transferred from program variables to a file.

2. Identify the unit. The unit identified by a data transfer I/O statement must be connected to a file when execution of the statement begins. Note that the file can be preconnected.

3. Establish the format, if one is specified. If specified, the format specifier is given in the data transfer statement and implies list-directed, namelist, or formatted data transfer.

4. Position the file prior to transferring the data. The position depends on the method of access (sequential or direct) and is described in subsection 9.3.2, page 413.

5. Transfer data between the file and the entities specified by the I/O item list (if any). The list items are processed in the order of the I/O list for all data transfer I/O statements, except namelist input data transfer statements, which are processed in the order of the entities specified within the input records. For namelist output data transfer, the output items are specified by the namelist when a namelist group name is used.

6. Determine if an error, end-of-record, or end-of-file condition exists. If one of these conditions occurs, the status of the file and the I/O items is as specified in subsection 9.4, page 414.

7. Position the file after transferring the data. The file position depends on whether one of the conditions in step 6 above occurred or if the data transfer was advancing or nonadvancing.

8. Cause the variables specified in the `IOSTAT=` and `SIZE=` specifiers, if present, to become defined. See the description of these specifiers in the `READ` and `WRITE` data transfer statements in subsection 9.2.2, page 383.

9. If `ERR=`, `END=`, or `EOR=` specifiers appear in the statement, transfer to the branch target corresponding to the condition that occurs. If an `IOSTAT=` specifier appears in the statement and the label specifier corresponding to the condition that occurs does not appear in the statement, the next statement in the execution sequence is executed. Otherwise, the execution of the program terminates. See the descriptions of these label specifiers in subsection 9.2.2, page 383.

Section 9

Data transfer
9.3.1

Data is transferred between records in the file and entities in the I/O list or namelist. The list items are processed in the order of the I/O list for all data transfer I/O statements except namelist data transfer statements. The list items for a namelist formatted data transfer input statement are processed in the order of the entities specified within the input records. The list items for a namelist data transfer output statement are processed in the order in which the data objects (variables) are specified in the namelist group object list.

The next item to be processed in the input or output item list is the *next effective item,* which is used to determine the interaction between the I/O item list and the format specification.

Zero-sized arrays and implied-`DO` lists with zero iteration counts are ignored in determining the next effective item.

Before beginning the I/O processing of a particular list item, the compiler first evaluates all values needed to determine which entities are specified by the list item. For example, the subscripts of a variable in an I/O list are evaluated before any data is transferred.

The value of an item that appears early in an I/O list can affect the processing of an item that appears later in the list. In the following example, the old value of `N` identifies the unit, but the new value of `N` is the subscript of `X`:

```
READ(N) N, X(N)
```

File position prior to data transfer
9.3.2

The file position prior to data transfer depends on the method of access: sequential or direct.

For sequential access on input, if there is a current record, the file position is not changed; this will be the case if the previous data transfer was nonadvancing. Otherwise, the file is positioned at the beginning of the next record and this record becomes the current record. Input must not occur if there is no next record (there must be an end-of-file record at least) or if there is a current record and the last data transfer statement accessing the file performed output.

If the file contains an end-of-file record, the file must not be positioned after the end-of-file record prior to data transfer. If you are using the CF90 system on a Cray PVP system or CRAY T3E system, multiple end-of-file records are permitted for some file structures. SPARC systems, do not allow multiple end-of-file records. However, a `REWIND` or `BACKSPACE` statement can be used to reposition the file.

> **ANSI/ISO:** The Fortran standard does not allow files that contain an end-of-file record to be positioned after the end-of-file record prior to data transfer .

For sequential access on output, if there is a current record, the file position is not changed; this will be the case if the previous data transfer was nonadvancing. Otherwise, a new record is created as the next record of the file; this new record becomes the last and current record of the file and the file is positioned at the beginning of this record.

For direct access, the file is positioned at the beginning of the record specified. This record becomes the current record.

File position after data transfer

9.3.3

If an error condition exists, the file position is indeterminate. If no error condition exists, but an end-of-file condition exists as a result of reading an end-of-file record, the file is positioned after the end-of-file record.

If no error condition or end-of-file condition exists, but an end-of-record condition exists, the file is positioned after the record just read. If no error condition, end-of-file condition, or end-of-record condition exists, and the data transfer was a nonadvancing input or output statement, the file position is not changed. In all other cases, the file is positioned after the record just read or written, and that record becomes the preceding record.

Error and other conditions in I/O statements

9.4

In step 6 of the execution model in subsection 9.3, page 411, the data transfer statements admit the occurrence of error and other conditions during the execution of the statement. The `OPEN`, `CLOSE`, `INQUIRE`, and file positioning statements also admit the occurrence of error conditions.

Whenever an error condition is detected, the variable of the `IOSTAT=` specifier is assigned a positive value, if it is present. Also, if the `ERR=` specifier is present, the program transfers to the branch target specified by the `ERR=` specifier upon completion of the I/O statement.

In addition, two other conditions might be detected: *end-of-file* and *end-of-record* . For each of these conditions, branches can be provided using the `END=` and `EOR=` specifiers in a `READ` statement to which the program branches upon completion of the `READ` statement. Also, the variable of the `IOSTAT=` specifier, if present, is set to a unique negative integer value, indicating which condition occurred.

An *end-of-file* condition occurs when either an end-of-file record is encountered during a sequential `READ` statement, or an attempt is made to read beyond the end of an internal file. An end-of-file condition can occur at the beginning of the execution of an input statement or during the execution of a formatted `READ` statement when more than one record is required by the interaction of the format specification and the input item list.

An *end-of-record* condition occurs when a nonadvancing input statement attempts to transfer data from beyond the end of the record.

Two or more conditions can occur during a single execution of an I/O statement. If one or more of the conditions is an error condition, one of the error conditions takes precedence, in the sense that the `IOSTAT=` specifier is given a positive value designating the particular error condition and the action taken by the I/O statement is as if only that error condition occurred.

In summary, an error condition can be generated by any of the I/O statements, and an end-of-file or end-of-record condition can be generated by a `READ` statement. The `IOSTAT=`, `END=`, `EOR=`, and `ERR=` specifiers allow the program to recover from such conditions rather than terminate execution of the program. In particular, when any one of these conditions occurs, the CF90 compiler takes the following actions:

1. If an end-of-record condition occurs and if the connection has been made with the `PAD=` specifier of `YES`, the record is padded, as necessary, with blanks to satisfy the input item list and the corresponding data edit descriptor . For the file position, see subsection 9.3.3, page 414.

2. Execution of the I/O statement terminates.

Section 9

3. If an error condition occurs, the position of the file becomes indeterminate; if an end-of-file condition occurs, the file is positioned after the end-of-file record; if an end-of-record condition occurs, the file is positioned after the current record.

4. If the statement also contains an `IOSTAT=` specifier, the variable specified becomes defined with a nonzero integer value; the value is positive if an error condition occurs and is negative if either an end-of-file or end-of-record condition occurs.

5. If the statement is a `READ` statement with a `SIZE=` specifier and an end-of-record condition occurs, then the variable specified by the `SIZE=` specifier becomes defined with an integer value equal to the number of characters read from the input record; blank padding characters inserted because the `PAD=` specifier is `YES` are not counted. For the file position, see subsection 9.3.3, page 414.

6. All implied-`DO` variables in the I/O statement become undefined; if an error or end-of-file condition occurs during execution of a `READ` statement, all list items become undefined; if an error condition occurs during the execution of an `INQUIRE` statement, all specifier variables except the `IOSTAT=` variable become undefined.

7. If an `END=` specifier is present and an end-of-file condition occurs, execution continues with the statement specified by the label in the `END=` specifier . If an `EOR=` specifier is present and an end-of-record condition occurs, execution continues with the statement specified by the label in the `EOR=` specifier. If an `ERR=` specifier is present and an error condition occurs, execution continues with the statement specified by the label in the `ERR=` specifier.

   If none of the preceding cases applies, but the I/O statement contains an `IOSTAT=` specifier, the normal execution sequence is resumed. If there is no `IOSTAT=`, `END=`, `EOR=`, or `ERR=` specifier, and an error or other condition occurs, the program terminates execution.

The following program segment illustrates how to handle end-of-file and error conditions:

```
READ (FMT = "(E8.3)", UNIT=3, IOSTAT=IOSS) X

IF (IOSS < 0) THEN

   ! PERFORM END-OF-FILE PROCESSING ON THE
   ! FILE CONNECTED TO UNIT 3.
   CALL END_PROCESSING

ELSE IF (IOSS > 0) THEN

   ! PERFORM ERROR PROCESSING
   CALL ERROR_PROCESSING

END IF
```

The procedure `END_PROCESSING` is used to handle the case where an end-of-file condition occurs and the procedure `ERROR_PROCESSING` is used to handle all other error conditions, because an end-of-record condition cannot occur.

`OPEN` statement

9.5

The `OPEN` statement establishes a connection between a unit and an external file and determines the connection properties. In order to perform data transfers (reading and writing), the file must be connected with an `OPEN` statement or be preconnected. It can also be used to change certain properties of the connection between the file and the unit, to create a file that is preconnected, or create a file and connect it.

The `OPEN` statement can appear anywhere in a program, and once executed, the connection of the unit to the file is valid in the main program or any subprogram for the remainder of that execution, unless a `CLOSE` statement affecting the connection is executed.

If a file is already connected to one unit, it must not be connected to a different unit.

Connecting a file to a unit
9.5.1

The `OPEN` statement connects an external file to a unit. If the file does not exist, it is created. If a unit is already connected to a file that exists, an `OPEN` statement referring to that unit may be executed. If the `FILE=` specifier is not included, the unit remains connected to the file. If the `FILE=` specifier names the same file, the `OPEN` statement may change the connection properties. If it specifies a different file by name, the effect is as if a `CLOSE` statement without a `STATUS=` specifier is executed on that unit and the `OPEN` statement is then executed. For information on default values for the `STATUS=` specifier, see subsection 9.6.1.3, page 431.

Creating a file on a preconnected unit
9.5.2

If a unit is preconnected to a file that does not exist, the `OPEN` statement creates the file and establishes properties of the connection.

Changing the connection properties
9.5.3

Execution of an `OPEN` statement can change the properties of a connection that is already established. The properties that can be changed are those indicated by `BLANK=`, `DELIM=`, `PAD=`, `ERR=`, and `IOSTAT=` specifiers. If new values for `DELIM=`, `PAD=`, and `BLANK=` specifiers are specified, these will be used in subsequent data transfer statements; otherwise, the old ones will be used. However, the values in `ERR=` and `IOSTAT=` specifiers, if present, apply only to the `OPEN` statement being executed; after that, the values of these specifiers have no effect. If no `ERR=` or `IOSTAT=` specifier appears in the new `OPEN` statement, error conditions will terminate the execution of the program.

Form of the `OPEN` statement

9.5.4

The `OPEN` statement and the connection specifier are defined as follows:

R904	*open_stmt*	**is**	`OPEN` (*connect_spec_list*)
R905	*connect_spec*	**is**	[`UNIT` =] *external_file_unit*
		or	`IOSTAT` = *scalar_default_int_variable*
		or	`ERR` = *label*
		or	`FILE` = *file_name_expr*
		or	`STATUS` = *scalar_char_expr*
		or	`ACCESS` = *scalar_char_expr*
		or	`FORM` = *scalar_char_expr*
		or	`RECL` = *scalar_int_expr*
		or	`BLANK` = *scalar_char_expr*
		or	`POSITION` = *scalar_char_expr*
		or	`ACTION` = *scalar_char_expr*
		or	`DELIM` = *scalar_char_expr*
		or	`PAD` = *scalar_char_expr*
R906	*file_name_expr*	**is**	*scalar_char_expr*

A unit specifier is required. If the keyword `UNIT` is omitted, the scalar integer expression must be the first item in the list. Note that the form `*` for the unit specifier is not allowed in the `OPEN` statement. However, in cases where the default external unit specified by an asterisk also corresponds to a nonnegative unit specifier (such as unit numbers `5` and `6` on many systems), inquiries about these default units and connection properties are possible.

A specifier must not appear more than once in an `OPEN` statement.

The character expression established for many of the specifiers must contain one of the permitted values from the list of alternative values for each specifier described in the following subsection. For example, `OLD`, `NEW`, `REPLACE`, `UNKNOWN`, or `SCRATCH` are permitted for the `STATUS=` specifier; any other combination of letters is not permitted. Trailing blanks in any specifier are ignored. The value specified can contain both uppercase and lowercase letters.

If the last data transfer to a unit connected for sequential access to a particular file is an output data transfer statement, an OPEN statement for that unit connecting it to a different file writes an end-of-file record to the original file.

9.5.5 The connection specifiers

The OPEN statement specifies the connection properties between the file and the unit, using keyword specifiers, which are described in this section. Table 26 indicates the possible values for the specifiers in an OPEN statement and their default values when the specifier is omitted.

Table 26. Values for keyword specifier variables in an OPEN statement

Specifier	Possible values	Default value
ACCESS=	DIRECT, SEQUENTIAL	SEQUENTIAL
ACTION=	READ, WRITE, READWRITE	File dependent
BLANK=	NULL, ZERO	NULL
DELIM=	APOSTROPHE, QUOTE, NONE	NONE
ERR=	Label	No default
FILE=	Character expression	fort. is prepended to the unit number
FORM=	FORMATTED UNFORMATTED	FORMATTED for sequential access UNFORMATTED for direct access
IOSTAT=	Scalar default integer variable	No default
PAD=	YES, NO	YES
POSITION=	ASIS, REWIND, APPEND	ASIS
RECL=	Positive scalar integer expression	File dependent

Section 9

Table 26. Values for keyword specifier variables in an `OPEN` statement (continued)

Specifier	Possible values	Default value
`STATUS=`	`OLD`, `NEW`, `UNKNOWN`, `REPLACE`, `SCRATCH`	`UNKNOWN`
`UNIT=`	Scalar integer expression	No default

9.5.5.1 `UNIT=` specifier

The `UNIT=` specifier has the following format:

```
[UNIT=] scalar_int_expr
```

The value of the *scalar_int_expr* must be nonnegative.

A unit specifier with an external unit is required. If the keyword `UNIT` is omitted, the unit specifier must be the first item in the list.

A unit number identifies one and only one external unit in all program units in a Fortran program.

9.5.5.2 `ACCESS=` specifier

The `ACCESS=` specifier has the following format:

```
ACCESS= scalar_char_expr
```

For *scalar_char_expr* , specify either `DIRECT` or `SEQUENTIAL`. `DIRECT` specifies the direct access method for data transfer. `SEQUENTIAL` specifies the sequential access method for data transfer; `SEQUENTIAL` is the default.

If the file exists, the method specified must be an allowed access method for the file.

Section 9

If the file is new, the allowed access methods given for the file must include the one indicated.

If the ACCESS= specifier has the value DIRECT, a RECL= specifier must be present.

ACTION= specifier
9.5.5.3

The ACTION= specifier has the following format:

```
ACTION= scalar_char_expr
```

Specify one of the following for *scalar_char_expr* :

Value	Function
READ	Indicates that WRITE, PRINT, and ENDFILE statements are prohibited
WRITE	Indicates that READ statements are prohibited
READWRITE	Indicates that any I/O statement is permitted

The default value depends on the file structure.

If READWRITE is an allowed ACTION= specifier, READ and WRITE must also be allowed ACTION= specifiers.

For an existing file, the specified action must be an allowed action for the file.

For a new file, the value of the ACTION= specifier must be one of the allowed actions for the file.

BLANK= specifier
9.5.5.4

The BLANK= specifier has the following format:

```
BLANK= scalar_char_expr
```

For *scalar_char_expr* , specify either NULL or ZERO. NULL specifies that the compiler should ignore all blanks in numeric fields; this is the default. ZERO specifies that the I/O library should interpret all blanks except leading blanks as zeros.

A field of all blanks evaluates to zero regardless of the argument specified to BLANK=.

The BLANK= specifier may be specified for files connected only for formatted I/O.

DELIM= specifier 9.5.5.5

The DELIM= specifier has the following format:

```
DELIM= scalar_char_expr
```

Specify one of the following for *scalar_char_variable* :

Value	Function
APOSTROPHE	Use the apostrophe as the delimiting character for character constants written by a list-directed or namelist formatted data transfer statement.
QUOTE	Use the quotation mark as the delimiting character for character constants written by a list-directed or namelist-formatted data transfer statement.
NONE	Use no delimiter to delimit character constants written by a list-directed or namelist-formatted data transfer statement. This is the default.

If the DELIM= specifier is APOSTROPHE, any occurrence of an apostrophe within a character constant will be doubled; if the DELIM= specifier is QUOTE, any occurrence of a quote within a character constant will be doubled.

The specifier is permitted only for a file connected for formatted I/O; it is ignored for formatted input.

`ERR=` specifier 9.5.5.6

The `ERR=` specifier has the following format:

```
ERR= label
```

The program branches to the label in the `ERR=` specifier if an error occurs in the `OPEN` statement. The label must be the label of a branch target statement in the same scoping unit as the `OPEN` statement.

If an error condition occurs, the position of the file becomes indeterminate.

If an `IOSTAT=` specifier is present and an error condition occurs, the `IOSTAT` variable specified becomes defined with a positive value.

`FILE=` specifier 9.5.5.7

The `FILE=` specifier has the following format:

```
FILE= scalar_char_expr
```

For *scalar_char_expr*, specify the name of the file to be connected. This is called the *file name expression.*

Trailing blanks in the file name are ignored.

The rules regarding file name lengths and path name lengths depend upon your platform, as follows:

- Cray PVP systems and CRAY T3E systems allow 256 characters for a file name. This file name can include a directory name and 1023 characters for the full path name.
- SPARC systems, allow 255 characters for a file name. The full path name is limited to 1023 characters.

Example:

```
FILENM = 'dir/fnam'
FNUM = '/subfnum'
OPEN (UNIT = 8, FILE = 'beauty')
OPEN (UNIT = 9, FILE = FILENM)
OPEN (UNIT = 10, FILE = FNAME//FNUM)
```

The `FILE=` specifier must appear if the `STATUS=` specifier is `OLD`, `NEW`, or `REPLACE`; the `FILE=` specifier must not appear if the `STATUS=` specifier is `SCRATCH`.

If the `FILE=` specifier is omitted and the unit is not already connected to a file, the `STATUS=` specifier must have the value `SCRATCH` and the unit becomes connected to a file with the unit number appended to the string `fort.`.

`FORM=` specifier 9.5.5.8

The `FORM=` specifier has the following format:

```
FORM= scalar_char_expr
```

For *scalar_char_expr* , specify either `FORMATTED` or `UNFORMATTED`. `FORMATTED` indicates that all records are formatted. `UNFORMATTED` indicates that all records are unformatted.

The default value is `UNFORMATTED` if the file is connected for direct access and the `FORM=` specifier is absent.

The default value is `FORMATTED` if the file is connected for sequential access and the `FORM=` specifier is absent.

If the file is new, the allowed forms given for the file must include the one indicated.

If the file exists, the form specified by the `FORM=` specifier must be one of the allowed forms for the file.

9.5.5.9 IOSTAT= specifier

The IOSTAT= specifier has the following format:

```
IOSTAT= scalar_default_int_variable
```

The I/O library will return an integer value in *scalar_default_int_variable*. If *scalar_default_int_variable* > 0, an error condition occurred. If *scalar_default_int_variable* = 0, no error condition occurred. Note that the value cannot be negative.

The IOSTAT= specifier applies to the execution of the OPEN statement itself.

9.5.5.10 PAD= specifier

The PAD= specifier has the following format:

```
PAD= scalar_char_expr
```

For *scalar_char_expr*, specify either YES or NO. YES indicates that the I/O library should use blank padding when the input item list and format specification require more data than the record contains; this is the default. NO indicates that the I/O library requires that the input record contain the data indicated by the input list and format specification.

The PAD= specifier is permitted only for a file connected for formatted I/O; it is ignored for formatted output.

If this specifier has the value YES and an end-of-record condition occurs, the data transfer behaves as if the record were padded with sufficient blanks to satisfy the input item and the corresponding data edit descriptor.

9.5.5.11 `POSITION=` specifier

The `POSITION=` specifier has the following format:

```
POSITION= scalar_char_expr
```

Specify one of the following for *scalar_char_expr* :

Value	Function
`ASIS`	Indicates that the file position is to remain unchanged for a connected file and is unspecified for a file that is not connected. `ASIS` permits an `OPEN` statement to change other connection properties of a file that is already connected without changing its position. This is the default.
`REWIND`	Indicates that the file is to be positioned at its initial point.
`APPEND`	Indicates that the file is to be positioned at the terminal point or just before an end-of-file record, if there is one.

The file must be connected for sequential access.

If the file is new, it is positioned at its initial point, regardless of the value of the `POSITION=` specifier.

9.5.5.12 `RECL=` specifier

The `RECL=` specifier has the following format:

```
RECL= scalar_int_expr
```

For *scalar_int_expr* , specify a positive value that specifies either the length of each direct access record or the maximum length of a sequential access record.

If the `RECL=` specifier is absent for a file connected for sequential access, the default value is 267 characters.

Section 9

The `RECL=` specifier must be present for a file connected for direct access.

If the file is connected for formatted I/O, the length is the number of characters.

If the file is connected for unformatted I/O, the length is measured in 8-bit bytes.

If the file exists, the length of the record specified must be an allowed record length.

If the file does not exist, the file is created with the specified length as an allowed length.

9.5.5.13 `STATUS=` specifier

The `STATUS=` specifier has the following format:

`STATUS=` *scalar_char_expr*

Specify one of the following for *scalar_char_expr* :

Value	Function
`OLD`	Requires that the file exist.
`NEW`	Requires that the file not exist.
`UNKNOWN`	Indicates that the file has an unknown status. This is the default.
`REPLACE`	Indicates that if the file does not exist, the file is created and given a status of `OLD`; if the file does exist, the file is deleted, a new file is created with the same name, and the file is given a status of `OLD`.
`SCRATCH`	Indicates that an unnamed file is to be created and connected to the specified unit; it is to exist either until the program terminates or a `CLOSE` statement is executed on that unit.

Scratch files must be unnamed; that is, the `STATUS=` specifier must not be `SCRATCH` when a `FILE=` specifier is present. The term *scratch file* refers to this temporary file.

Note that if the `STATUS=` specifier is `REPLACE`, the specifier in this statement is not changed to `OLD`; only the file status is considered to be `OLD` when the file is used in subsequently executed I/O statements, such as a `CLOSE` statement.

`CLOSE` statement

9.6

Executing a `CLOSE` statement terminates the connection between a file and a unit. Any connections not closed explicitly by a `CLOSE` statement are closed by the operating system when the program terminates, unless an error condition has terminated the program.

The `CLOSE` statement is defined as follows:

R907	*close_stmt*	**is**	`CLOSE (` *close_spec_list* `)`
R908	*close_spec*	**or**	`STATUS` = *scalar_char_expr*
		is	[`UNIT` =] *external_file_unit*
		or	`IOSTAT` = *scalar_default_int_variable*
		or	`ERR` = *label*

A specifier must not appear more than once in a `CLOSE` statement.

A `CLOSE` statement may appear in any program unit in an executing program.

A `CLOSE` statement may refer to a unit that is not connected or does not exist, but it has no effect.

If the last data transfer to a file connected for sequential access is an output data transfer statement, a `CLOSE` statement for a unit connected to this file writes an end-of-file record to the file.

After a unit has been disconnected by a CLOSE statement, it can be connected again to the same or a different file. Similarly, after a file has been disconnected by a CLOSE statement, it can also be connected to the same or a different unit, provided the file still exists.

```
CLOSE (ERR=99, UNIT=9)
CLOSE (8, IOSTAT=IR, STATUS="KEEP")
```

The subsections that follow describe the components of the *close_spec_list* in more detail.

9.6.1 UNIT= specifier

The UNIT= specifier has the following format:

[UNIT=] *scalar_int_expr*

The value of the *scalar_int_expr* must be nonnegative.

A unit specifier is required. If the keyword UNIT is omitted, *scalar_int_expr* must be the first item in the list.

A unit number identifies one and only one external unit in all program units in a Fortran program.

9.6.1.1 ERR= specifier

The ERR= specifier has the following format:

ERR= *label*

If an error condition occurs, the position of the file becomes indeterminate.

If an IOSTAT= specifier is present and an error condition occurs, the IOSTAT variable specified becomes defined with a positive value.

The program branches to the label in the ERR= specifier if an error occurs in the CLOSE statement. The label must be the label of a branch target statement in the same scoping unit as the CLOSE statement.

9.6.1.2 IOSTAT= specifier

The format of the IOSTAT= specifier is as follows:

```
IOSTAT= scalar_default_int_variable
```

The I/O library returns an integer value in *scalar_default_int_variable*. If *scalar_default_int_variable* > 0, an error condition occurred. If *scalar_default_int_variable* = 0, no error condition occurred. Note that the value cannot be negative.

The IOSTAT= specifier applies to the execution of the CLOSE statement itself.

9.6.1.3 STATUS= specifier

The format of the STATUS= specifier is as follows:

```
STATUS= scalar_char_expr
```

For *scalar_char_expr* , specify either KEEP or DELETE. KEEP indicates that the file is to continue to exist after closing the file. DELETE indicates that the file will not exist after closing the file.

If the unit has been opened with a STATUS= specifier of SCRATCH, the default is DELETE. If the unit has been opened with any other value of the STATUS= specifier, the default is KEEP.

KEEP must not be specified for a file whose file status is SCRATCH.

If KEEP is specified for a file that does not exist, the file does not exist after the CLOSE statement is executed.

Inquiring about files

9.7

An inquiry can be made about a file's existence, connection, access method, or other properties. For each property inquired about, a scalar variable of default kind must be supplied; that variable is given a value that answers the inquiry. The variable can be tested and optional execution paths can be selected based on the answer returned. The inquiry specifiers are determined by keywords in the `INQUIRE` statement. The only exception is the unit specifier, which, if no keyword is specified, must be the first specifier . A file inquiry can be made by unit number, file name, or an output item list. When inquiring by an output item list, an output item list that might be used in an unformatted direct access output statement must be present.

The `INQUIRE` statement

9.7.1

There are three kinds of `INQUIRE` statements: inquiry by unit, by name, and by an output item list. The first two kinds use the *inquire_spec_list* form of the `INQUIRE` statement. The third kind uses the `IOLENGTH` form. Inquiry by unit uses a unit specifier. Inquiry by file uses a file specifier with the keyword `FILE=`. These statements are defined as follows:

R923	*inquire_stmt*	**is**	`INQUIRE (` *inquire_spec_list* `)`
		or	`INQUIRE ( IOLENGTH =` *scalar_default_int_variable* `)` ≡
			≡ *output_item_list*
R924	*inquire_spec*	**is**	[`UNIT =`] *external_file_unit*
		or	`FILE =` *file_name_expr*
		or	`IOSTAT =` *scalar_default_int_variable*
		or	`ERR =` *label*
		or	`EXIST =` *scalar_default_logical_variable*
		or	`OPENED =` *scalar_default_logical_variable*
		or	`NUMBER =` *scalar_default_int_variable*
		or	`NAMED =` *scalar_default_logical_variable*
		or	`NAME =` *scalar_char_variable*
		or	`SEQUENTIAL =` *scalar_char_variable*
		or	`ACCESS =` *scalar_char_variable*
		or	`DIRECT =` *scalar_char_variable*
		or	`FORM =` *scalar_char_variable*
		or	`FORMATTED =` *scalar_char_variable*
		or	`UNFORMATTED =` *scalar_char_variable*
		or	`RECL =` *scalar_default_int_variable*
		or	`NEXTREC =` *scalar_default_int_variable*
		or	`BLANK =` *scalar_char_variable*
		or	`POSITION =` *scalar_char_variable*
		or	`ACTION =` *scalar_char_variable*
		or	`READ =` *scalar_char_variable*
		or	`WRITE =` *scalar_char_variable*
		or	`READWRITE =` *scalar_char_variable*
		or	`DELIM =` *scalar_char_variable*
		or	`PAD =` *scalar_char_variable*

Section 9

An `INQUIRE` statement with an *inquire_spec_list* must have a unit specifier or a `FILE=` specifier, but not both. If the keyword `UNIT` is omitted, a scalar integer expression must be the first item in the list and must have a nonnegative value.

No specifier can appear more than once in a given inquiry specifier list.

For an inquiry by an output item list, the output item list must be a valid output list for an unformatted, direct-access, output statement. The length value returned in the scalar default integer variable must be a value that is acceptable when used as the value of the RECL= specifier in an OPEN statement. This value can be used in a RECL= specifier to connect a file whose records will hold the data indicated by the output list of the INQUIRE statement.

An INQUIRE statement can be executed before or after a file is connected to a unit. The specifier values returned by the INQUIRE statement are those current at the time at which the INQUIRE statement is executed.

A variable appearing in a specifier or any entity associated with it must not appear in another specifier in the same INQUIRE statement if that variable can become defined or undefined as a result of executing the INQUIRE statement. That is, do not try to assign two inquiry results to the same variable.

Except for the NAME= specifier, character values are returned in uppercase.

If an error condition occurs during the execution of an INQUIRE statement, all the inquiry specifier variables become undefined except the IOSTAT= specifier .

```
INQUIRE (9, EXIST = EX)
INQUIRE (FILE = ”T123”, OPENED = OP, ACCESS = AC)
INQUIRE (IOLENGTH = IOLEN)  X, Y, CAT
```

Section 9

Specifiers for inquiry by unit or file name
9.7.2

This subsection describes the format and effect of the inquiry specifiers that may appear in the inquiry by unit and file forms of the INQUIRE statement.

UNIT= specifier
9.7.2.1

The format of the UNIT= specifier is as follows:

[UNIT=] *scalar_int_expr*

The *scalar_int_expr* indicates an external unit (R902). The value of the *scalar_int_expr* must be nonnegative.

To inquire by unit, a unit specifier must be present, and a file specifier cannot be present. If the keyword `UNIT` is omitted, a *scalar_int_expr* must be the first item in the list.

A unit number identifies one and only one external unit in all program units in a Fortran program.

The file is the file connected to the unit, if one is connected; otherwise, the file does not exist.

`ACCESS=` specifier 9.7.2.2

The format of the `ACCESS=` specifier is as follows:

```
ACCESS= scalar_char_variable
```

One of the following is returned in *scalar_char_variable* :

Value	Function
`SEQUENTIAL`	Indicates that the file is connected for sequential access
`DIRECT`	Indicates that the file is connected for direct access
`UNDEFINED`	Indicates that the file is not connected

`ACTION=` specifier 9.7.2.3

The format of the `ACTION=` specifier is as follows:

```
ACTION= scalar_char_variable
```

One of the following is returned in *scalar_char_variable* :

Value	Function
READ	Indicates that the file is connected with access limited to input only
WRITE	Indicates that the file is connected with access limited to output only
READWRITE	Indicates that the file is connected for both input and output
UNDEFINED	Indicates that the file is not connected

9.7.2.4 BLANK= specifier

The format of the BLANK= specifier is as follows:

```
BLANK= scalar_char_variable
```

One of the following is returned in *scalar_char_variable* :

Value	Function
NULL	Indicates that null blank control is in effect
ZERO	Indicates that zero blank control is in effect
UNDEFINED	Indicates that the file is not connected for formatted I/O or the file is not connected at all

See the BLANK= specifier for the OPEN statement in subsection 9.5.5, page 420 for the meaning of null and zero blank control.

Section 9

`DELIM=` specifier 9.7.2.5

The format of the `DELIM=` specifier is as follows:

```
DELIM= scalar_char_variable
```

One of the following is returned in *scalar_char_variable* :

Value	Function
`APOSTROPHE`	Indicates that an apostrophe is used as the delimiter in list-directed and namelist-formatted output
`QUOTE`	Indicates that the quotation mark is used as the delimiter in list-directed and namelist-formatted output
`NONE`	Indicates that there is no delimiting character in list-directed and namelist-formatted output
`UNDEFINED`	Indicates that the file is not connected or the file is not connected for formatted I/O

`DIRECT=` specifier 9.7.2.6

The format of the `DIRECT=` specifier is as follows:

```
DIRECT= scalar_char_variable
```

One of the following is returned in *scalar_char_variable* :

Value	Function
`YES`	Indicates that direct access is an allowed access method
`NO`	Indicates that direct access is not an allowed access method
`UNKNOWN`	Indicates that the system does not know if direct access is allowed

9.7.2.7 ERR= specifier

The format of the ERR= specifier is as follows:

```
ERR= label
```

The program branches to the label in the ERR= specifier if there is an error in the execution of the INQUIRE statement itself. The label must be the label of a branch target statement in the same scoping unit as the INQUIRE statement.

If an error condition occurs, the position of the file becomes indeterminate.

If an IOSTAT= specifier is present and an error condition occurs, the IOSTAT variable specified becomes defined with a positive value. All other inquiry specifier variables become undefined.

9.7.2.8 EXIST= specifier

The format of the EXIST= specifier is as follows:

```
EXIST= scalar_default_logical_variable
```

The system returns either .TRUE. or .FALSE. in *scalar_default_logical_variable* . .TRUE. indicates that the file or unit exists. .FALSE. indicates that the file or unit does not exist.

9.7.2.9 FILE= specifier

The format of the FILE= specifier is as follows:

```
FILE= scalar_char_expr
```

To inquire by file, a file specifier must be present, and a unit specifier cannot be present.

The value of the *scalar_char_expr* must be an acceptable file name. Trailing blanks are ignored. Both uppercase and lowercase letters are acceptable. The use of uppercase and lowercase is significant for file names.

The file name can refer to a file not connected or to one that does not exist.

9.7.2.10 `FORM=` specifier

The format of the `FORM=` specifier is as follows:

`FORM=` *scalar_char_variable*

The system returns one of the following in *scalar_char_variable* :

Value	Function
`FORMATTED`	Indicates that the file is connected for formatted I/O
`UNFORMATTED`	Indicates that the file is connected for unformatted I/O
`UNDEFINED`	Indicates that the file is not connected

9.7.2.11 `FORMATTED=` specifier

The format of the `FORMATTED=` specifier is as follows:

`FORMATTED=` *scalar_char_variable*

The system returns one of the following in *scalar_char_variable* :

Value	Function
`YES`	Indicates that formatted I/O is an allowed form for the file
`NO`	Indicates that formatted I/O is not an allowed form for the file
`UNKNOWN`	Indicates that the system cannot determine if formatted I/O is an allowed form for the file

9.7.2.12 IOSTAT= specifier

The format of the IOSTAT= specifier is as follows:

```
IOSTAT= scalar_default_int_variable
```

The I/O library returns an integer value in *scalar_default_int_variable*. If *scalar_default_int_variable* > 0, an error condition occurred. If *scalar_default_int_variable* = 0, no error condition occurred. Note that the value cannot be negative.

The IOSTAT= specifier applies to the execution of the INQUIRE statement itself.

9.7.2.13 NAME= specifier

The format of the NAME= specifier is as follows:

```
NAME= scalar_char_variable
```

The I/O library returns either a file name or an undefined value in *scalar_char_variable*. If the value is a file name, it is the name of the file connected to the unit, if the file has a name. An undefined value indicates that either file does not have a name or no file is connected to the unit.

A name different from the one specified in the FILE= specifier on the OPEN statement may be returned.

9.7.2.14 NAMED= specifier

The format of the NAMED= specifier is as follows:

```
NAMED= scalar_default_logical_variable
```

The system returns either `.TRUE.` or `.FALSE.` in *scalar_default_logical_variable* . `.TRUE.` indicates that the file has a name. `.FALSE.` indicates that the file does not have a name.

9.7.2.15 `NEXTREC=` specifier

The format of the `NEXTREC=` specifier is as follows:

`NEXTREC=` *scalar_default_int_variable*

The system returns one of the following in *scalar_default_int_variable* :

Value	Function
last record number + `1`	Indicates the next record number to be read or written in a file connected for direct access. The value is one more than the last record number read or written.
`1`	Indicates that no records have been processed.
undefined value	Indicates that the file is not connected for direct access or the file position is indeterminate because of a previous error condition.

This inquiry is used for files connected for direct access.

9.7.2.16 `NUMBER=` specifier

The format of the `NUMBER=` specifier is as follows:

`NUMBER=` *scalar_default_int_variable*

The system returns either a unit number or –1 in *scalar_default_int_variable* , . A unit number indicates the number of the unit connected to the file. –1 indicates that there is no unit connected to the file.

Section 9

9.7.2.17 `OPENED=` specifier

The format of the `OPENED=` specifier is as follows:

```
OPENED= scalar_default_logical_variable
```

The system returns either `.TRUE.` or `.FALSE.` in *scalar_default_logical_variable*. `.TRUE.` indicates that the file or unit is connected (that is, opened). `.FALSE.` indicates that the file or unit is not connected (that is, not opened).

9.7.2.18 `PAD=` specifier

The format of the `PAD=` specifier is as follows:

```
PAD= scalar_char_variable
```

The system returns either `NO` or `YES` in *scalar_char_variable*. `NO` indicates that the file or unit is connected with the `PAD=` specifier set to `NO`. `YES` indicates that either the file or unit is connected with the `PAD=` specifier set to `YES` or that the file or unit is not connected.

9.7.2.19 `POSITION=` specifier

The format of the `POSITION=` specifier is as follows:

```
POSITION= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable* :

Value	Function
`REWIND`	Indicates that the file is connected with its position at the initial point
`APPEND`	Indicates that the file is connected with its position at the terminal point
`ASIS`	Indicates that the file is connected without changing its position
`UNDEFINED`	Indicates that either the file is not connected or it is connected for direct access

If any repositioning has occurred since the file was connected, the value returned is `ASIS`. It is not equal to `REWIND` unless positioned at the initial point, and it is not equal to `APPEND` unless positioned at the terminal point.

9.7.2.20 `READ=` specifier

The format of the `READ=` specifier is as follows:

```
READ= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable* :

Value	Function
`YES`	Indicates that `READ` is one of the allowed actions for the file
`NO`	Indicates that `READ` is not one of the allowed actions for the file
`UNKNOWN`	Indicates that the action for the file cannot be determined

Section 9

9.7.2.21 READWRITE= specifier

The format of the READWRITE= specifier is as follows:

```
READWRITE= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable* :

Value	Function
YES	Indicates that READWRITE is an allowed action for the file
NO	Indicates that READWRITE is not an allowed action for the file
UNKNOWN	Indicates that the action for the file cannot be determined

9.7.2.22 RECL= specifier

The format of the RECL= specifier is as follows:

```
RECL= scalar_default_int_variable
```

The system returns either a maximum record length or an undefined value in *scalar_default_int_variable* . The maximum record length is the record length, if the file is connected for sequential access, or the length of each record, if the file is connected for direct access. An undefined value indicates that the file does not exist.

For a formatted file, the length is the number of characters for all records.

For an unformatted file, the length is in 8-bit bytes.

9.7.2.23 `SEQUENTIAL=` specifier

The format of the `SEQUENTIAL=` specifier is as follows:

> `SEQUENTIAL=` *scalar_char_variable*

The system returns one of the following in *scalar_char_variable* :

Value	Function
`YES`	Indicates that sequential access is an allowed access method
`NO`	Indicates that sequential access is not an allowed access method
`UNKNOWN`	Indicates that the access is not known

9.7.2.24 `UNFORMATTED=` specifier

The format of the `UNFORMATTED=` specifier is as follows:

> `UNFORMATTED=` *scalar_char_variable*

The system returns one of the following in *scalar_char_variable* :

Value	Function
`YES`	Indicates that unformatted I/O is an allowed form for the file
`NO`	Indicates that unformatted I/O is not an allowed form for the file
`UNKNOWN`	Indicates that the form cannot be determined

WRITE= specifier 9.7.2.25

The format of the WRITE= specifier is as follows:

```
WRITE= scalar_char_variable
```

The system returns one of the following in *scalar_char_variable* :

Value	Function
YES	Indicates that WRITE is an allowed action for the file
NO	Indicates that WRITE is not an allowed action for the file
UNKNOWN	Indicates that the action cannot be determined

Table of values assigned by the INQUIRE statement 9.7.3

Table 27 summarizes the values assigned to the INQUIRE specifier variables by the execution of an INQUIRE statement.

Table 27. Values for specifier variables in an INQUIRE statement

	INQUIRE by file		INQUIRE by unit	
Specifier	Unconnected	Connected	Connected	Unconnected
ACCESS=	UNDEFINED	SEQUENTIAL or DIRECT		UNDEFINED
ACTION=	UNDEFINED	READ, WRITE, or READWRITE		UNDEFINED
BLANK=	UNDEFINED	NULL, ZERO, or UNDEFINED		UNDEFINED
DELIM=	UNDEFINED	APOSTROPHE, QUOTE, NONE, or UNDEFINED		UNDEFINED
DIRECT=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN

Table 27. Values for specifier variables in an INQUIRE statement (continued)

Specifier	INQUIRE by file: Unconnected	INQUIRE by file: Connected	INQUIRE by unit: Connected	INQUIRE by unit: Unconnected
EXIST=	.TRUE. if file exists, .FALSE. otherwise		.TRUE. if unit exists, .FALSE. otherwise	
FORM=	UNDEFINED	FORMATTED or UNFORMATTED		UNDEFINED
FORMATTED=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
IOSTAT=	0 for no error, a positive integer for an error			
NAME=	File name (may not be same as FILE= value)	File name if named, else undefined		Undefined
NAMED=	.TRUE.	.TRUE. if file named, .FALSE. otherwise		.FALSE.
NEXTREC=	Undefined	If direct access, next record number; else undefined		Undefined
NUMBER=	-1	Unit number		-1
OPENED=	.FALSE.	.TRUE.		.FALSE.
PAD=	YES	YES or NO		YES
POSITION=	UNDEFINED	REWIND, APPEND, ASIS, or UNDEFINED		UNDEFINED
READ=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
READWRITE=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
RECL=	Undefined	If direct access, record length; else maximum record length		Undefined
SEQUENTIAL=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN

Section 9

Table 27. Values for specifier variables in an INQUIRE statement (continued)

Specifier	INQUIRE by file		INQUIRE by unit	
	Unconnected	Connected	Connected	Unconnected
UNFORMATTED=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
WRITE=	UNKNOWN	YES, NO, or UNKNOWN		UNKNOWN
IOLENGTH=	RECL= value for *output_item_list*			

File positioning statements

9.8

Execution of a data transfer statement usually changes the file position. In addition, there are three statements whose main purpose is to change the file position. Changing the position backwards by one record is called *backspacing* and is performed by the BACKSPACE statement. Changing the position to the beginning of the file is called *rewinding* and is performed by the REWIND statement. The ENDFILE statement writes an end-of-file record and positions the file after the end-of-file record.

```
BACKSPACE 9
BACKSPACE (UNIT = 10)
BACKSPACE (ERR = 99, UNIT = 8, IOSTAT = STATUS)
REWIND (ERR = 102, UNIT = 10)
ENDFILE (10, IOSTAT = IERR)
ENDFILE (11)
```

The file positioning statements have common components.

- The *external_file_unit* can be a unit specifier or an external name. If it is a unit specifier, it must have a nonnegative value. See subsection 9.1.5, page 371, for information on units and the *external_file_unit* .
- The *position_spec_list* is described in subsection 9.8.4, page 452.

ANSI/ISO: The Fortran 90 standard does not address using external names as *external_file_unit* specifiers.

The following subsections describe the BACKSPACE, REWIND, and ENDFILE statements in more detail.

BACKSPACE statement
9.8.1

Executing a BACKSPACE statement causes the file to be positioned before the current record if there is a current record, or before the preceding record if there is no current record. If there is no current record and no preceding record, the file position is not changed. If the preceding record is an end-of-file record, the file becomes positioned before the end-of-file record.

If the last data transfer to a file connected for sequential access was an output data transfer statement, a BACKSPACE statement on that file writes an end-of-file record to the file. If there is a preceding record, the BACKSPACE statement causes the file to become positioned before the record that precedes the end-of-file record.

The BACKSPACE statement is defined as follows:

R919	*backspace_stmt*	**is**	BACKSPACE *external_file_unit*
		or	BACKSPACE (*position_spec_list*)

The BACKSPACE statement is used only to position external files.

If the file is already at its initial point, a BACKSPACE statement has no effect. If the file is connected, but does not exist, backspacing is prohibited. Backspacing over records written using list-directed or namelist formatting is prohibited.

```
BACKSPACE  ERROR_UNIT         ! ERROR_UNIT is an
                              !    integer variable
BACKSPACE (10, &              ! STAT is an integer
                              !    variable of
           IOSTAT = STAT)     !    default type
```

Section 9

The REWIND statement
9.8.2

A REWIND statement positions the file at its initial point. Rewinding has no effect on the file position when the file is already positioned at its initial point. If a file does not exist, but it is connected, rewinding the file is permitted, but it has no effect.

If the last data transfer to a file was an output data transfer statement, a REWIND statement on that file writes an end-of-file record to the file.

The file must be connected for sequential access.

The REWIND statement is defined as follows:

R921	*rewind_stmt*	**is**	REWIND *external_file_unit*
		or	REWIND (*position_spec_list*)

The REWIND statement is used only to position external files.

```
REWIND  INPUT_UNIT       ! INPUT_UNIT is an integer
                         !   variable
REWIND (10, ERR = 200)   ! 200 is a label of branch
                         !   target in this
                         !   scoping unit
```


The ENDFILE statement
9.8.3

The ENDFILE writes an end-of-file record as the next record and positions the file after the end-of-file record written. Writing records past the end-of-file record is prohibited, except for certain file structures.

If you are using the CF90 compiler on a Cray PVP system or a CRAY T3E system, you can find more information on file structures in the *Application Programmer's I/O Guide*, publication SG–2168.

If you are using the CF90 compiler on a SPARC system, UNIX record blocking is the default form for sequential unformatted files. Text files are used for all other types of Fortran I/O. Text files are described in the *Application Programmer's I/O Guide*, publication SG–2168.

ANSI/ISO: The Fortran 90 standard does not address writing past an end-of-file record.

After executing an `ENDFILE` statement, it is necessary to execute a `BACKSPACE` or `REWIND` statement to position the file ahead of the end-of-file record before reading or writing the file. If the file is connected but does not exist, writing an end-of-file record creates the file.

The file must be connected for sequential access.

The `ENDFILE` statement is defined as follows:

R920	*endfile_stmt*	**is**	`ENDFILE` *external_file_unit*
		or	`ENDFILE (` *position_spec_list* `)`

The `ENDFILE` statement is used only to position external files.

```
                          ! OUTPUT_UNIT is an integer variable.
ENDFILE  OUTPUT_UNIT
ENDFILE (10, ERR = 200, IOSTAT = ST)
                          !   200 is a label of a branch
                          !   target in this scoping unit
                          !   ST is a default scalar integer
                          !   variable
```

A file can be connected for sequential and direct access, but not for both simultaneously. If a file is connected for sequential access and an `ENDFILE` statement is executed on the file, only those records written before the `ENDFILE` statement is executed are considered to have been written. Consequently, if the file is subsequently connected for direct access, only those records before the end-of-file record can be read.

Section 9

Specifiers for file position statements

9.8.4

The *position_spec_list* is defined as follows (note that no specifier can be repeated in a *position_spec_list*):

R922	*position_spec*	**is**	[`UNIT` =] *scalar_int_expr*
		or	`IOSTAT` = *scalar_default_int_variable*
		or	`ERR` = *label*

The following subsections describe the form and effect of the position specifiers that can appear in the file positioning statements.

`UNIT=` specifier

9.8.4.1

The format of the `UNIT=` specifier is as follows:

`[UNIT=]` *scalar_int_expr*

The *scalar_int_expr* specifies an external unit (R902). The value of the *scalar_int_expr* must be nonnegative.

A unit specifier is required. If the keyword `UNIT` is omitted, the scalar integer expression must be the first item in the position specifier list. A unit number identifies one and only one external unit in all program units in a Fortran program.

There must be a file connected to the unit, and the unit must be connected for sequential access.

Section 9

`IOSTAT=` specifier

9.8.4.2

The format of the `IOSTAT=` specifier is as follows:

`IOSTAT=` *scalar_default_int_variable*

The I/O library returns an integer value in *scalar_default_int_variable* . If *scalar_default_int_variable* > 0, an error condition occurred. If *scalar_default_int_variable* = 0, no error condition occurred. Note that the value cannot be negative.

The `IOSTAT=` specifier applies to the execution of the file positioning statement itself.

`ERR=` specifier
9.8.4.3

The format of the `ERR=` specifier is as follows:

`ERR=` *label*

The program branches to the label in the `ERR=` specifier if there is an error in the execution of the particular file positioning statement itself. The label must be the label of a branch target statement in the same scoping unit as the file positioning statement.

If an error condition occurs, the position of the file becomes indeterminate.

If an `IOSTAT=` specifier is present and an error condition occurs, the `IOSTAT` variable specified becomes defined with a positive value.

Restrictions on I/O specifiers, list items, and statements
9.9

Any function reference appearing in an I/O statement specifier or in an I/O list must not cause the execution of another I/O statement on the same file or unit. Note that such function references also must not have side effects that change any object in the same statement.

```
WRITE (10, FMT = ”(10I5)”, REC = FCN(I)) &
      X(FCN(J)), I, J
```

The function `FCN` must not contain an I/O statement using unit 10 and must not change its argument because `I` and `J` are also output list items.

Section 9

A unit or file might not have all of the properties (for example, all access methods or all forms) required for it by execution of certain I/O statements. If this is the case, such I/O statements must not refer to files or units limited in this way. For example, if unit 5 cannot support unformatted sequential files, the following OPEN statement must not appear in a program:

```
OPEN (UNIT = 5, IOSTAT = IERR,  &
      ACCESS = "SEQUENTIAL", FORM = "UNFORMATTED")
```

Input and Output (I/O) Editing [10]

Usually , data is stored in memory as the values of variables in some binary form. On the other hand, formatted data records in a file consist of characters. Thus, when data is read from a formatted record, it must be converted from characters to the internal representation. When data is written to a formatted record, it must be converted from the internal representation into a string of characters.

A *format specification* provides the information necessary to determine how these conversions are to be performed. The format specification is basically a list of *edit descriptors,* of which there are three general types: data edit descriptors, control edit descriptors, and string edit descriptors. There is a data edit descriptor for each data value in the input/output (I/O) list of the data transfer statement. Control edit descriptors specify the spacing and position within a record, new records, interpretation of blanks, and plus sign suppression. String edit descriptors transfer strings of characters represented in format specifications to output records.

The format reference that indicates where to find the format can be a statement label that identifies a `FORMAT` statement, or it can be a character expression giving the format directly. Using either method is called *explicit formatting.*

There are two other cases where formatting of a different sort applies. These are list-directed and namelist formatting. Formatting (that is, conversion) occurs without specifically providing the editing information usually contained in a format specification. In these cases, the editing or formatting is implicit; that is, the details about the width of fields, forms of output values, and location of output fields within the records is determined by the type of each data value in the I/O list.

This section describes the following information:

- The two methods of specifying explicit formatting.
- The three general types of edit descriptors and an algorithm that describes the correspondence of the data edit descriptors and items in the data item list of the data transfer statement.
- The two methods of implicit formatting: list-directed and namelist.

Table 28, Table 29, and Table 30 list the control, data, and string edit descriptors and provide a brief description of each.

Table 28. Summary of control edit descriptors

Descriptor	Description
`BN`	Ignore nonleading blanks in numeric input fields
`BZ`	Treat nonleading blanks in numeric input fields as zeros
`S`	Do not print optional plus sign
`SP`	Print plus sign
`SS`	Do not print plus sign
`T`	Tab to specified position
`TL`	Tab left the specified number of positions
`TR`	Tab right the specified number of positions
`X`	Tab right the specified number of positions
`$`	Suppress carriage control
`/`	End current record and move to beginning of next record
`:`	Stop format processing if no further input/output list items exist
`P`	Interpret certain real numbers with a specified scale factor

ANSI/ISO: The Fortran 90 standard does not specify the `$` control edit descriptor .

Table 29. Summary of data edit descriptors

Descriptor	Description
`A`	Convert data of type character
`B`	Convert data of type integer to/from a binary base
`D`	Convert data of type real; same as `E` edit descriptor
`E`	Convert data of type real with an exponent
`EN`	Convert data of type real to engineering notation
`ES`	Convert data of type real to scientific notation
`F`	Convert data of type real with no exponent on output
`G`	Convert data of all intrinsic types
`I`	Convert data of type integer
`L`	Convert data of type logical
`O`	Convert data of type integer to/from an octal base
`Z`	Convert data of type integer to/from a hexadecimal base

Table 30. Summary of string edit descriptors

Descriptor	Description
`H`	Transfer of text character to output record
'*text*'	Transfer of a character literal constant to output record
"*text*"	Transfer of a character literal constant to output record

The CF90 compiler extends the use of `B`, `O`, and `Z` edit descriptors to type complex (`KIND=8`), logical, and real (`KIND=8`).

Table 31 and Table 32 show the usage of the CF90 edit descriptors with all intrinsic data types. In these tables:

- *NA* indicates illegal usage that is not allowed.
- *I, O* indicates legal usage for input and output.
- *O* indicates legal usage for output.

Table 31. Edit descriptors with data types

Data types	I	F	E	D	G	L	A	B	O	Z	R
Character	NA	NA	NA	NA	NA	NA	I, O	NA	NA	NA	NA
Complex	NA	I, O	I, O	I, O	I, O	NA	I, O	I, O	I, O	I, O	I, O
Double precision	NA	I, O	I, O	I, O	I, O	NA	NA	NA	O	NA	NA
Integer	I, O	NA	NA	NA	NA	NA	I, O	I, O	I, O	I, O	I, O
Logical	NA	NA	NA	NA	NA	I, O	I, O	I, O	I, O	I, O	I, O
Real	NA	I, O	I, O	I, O	I, O	NA	I, O	I, O	I, O	I, O	I, O

Table 32. Descriptors and data types when SEGLDR or MPPLDR `EQUIV` directive is used (Cray PVP systems and CRAY T3E systems)

Data types	I	F	E	D	G	L	A	B	O	Z	R
Integer	I, O	I, O	I, O	NA	I, O	I, O	I, O	I, O	I, O	I, O	I, O
Logical	I, O	I, O	I, O	NA	I, O	I, O	I, O	I, O	I, O	I, O	I, O
Real	I, O	I, O	I, O	I, O	I, O	I, O	I, O	I, O	I, O	I, O	I, O

Table 32 shows the format restrictions for integer, logical, and real variables that are valid when you use SEGLDR or MPPLDR and the `EQUIV` directive. To change the limitations for read and write operations, specify `EQUIV=$RNOCHK($RCHK)` or `EQUIV=$WNOCHK($WCHK)`, respectively. Both of these `EQUIV` statements must be specified if changes are desired.

Note: SEGLDR and MPPLDR are not supported on SPARC systems.

Explicit formatting

10.1

Explicit formatting information can be provided in one of the following ways:

- Contained in a `FORMAT` statement, as follows:

```
     WRITE (6, 100) LIGHT, AND, HEAVY
 100 FORMAT (F10.2, I5, E16.8)
```

- Given as the value of a character expression, as follows:

```
 WRITE (6, '(F10.2, I5, E16.8)' ) LIGHT, AND, HEAVY
```

`FORMAT` statement

10.1.1

The `FORMAT` statement is defined as follows:

R1001	*format_stmt*	**is**	`FORMAT` *format_specification*
R1002	*format_specification*	**is**	([*format_item_list*])

As the following sample format shows, the `FORMAT` statement must be labeled. The label is used in the I/O statement to reference a particular `FORMAT` statement.

label `FORMAT ( [` *format_item_list* `] )`

Section 10

Each I/O statement in a program can reference a unique FORMAT statement, or a FORMAT statement can be used in more than one I/O statement.

Character expression format specifications
10.1.2

A character expression can be used in the I/O statement as a format specification. The leading part of the character expression must be a valid format specification including the parentheses; that is, the value of the expression must be such that the first nonblank character is a left parenthesis, followed by a list of valid format items, followed by a right parenthesis.

All variables in the character expression must be defined when the I/O statement is executed.

Characters can appear following the last right parenthesis in the character expression; they have no effect.

If the expression is a character array, the format is scanned in array element order. For example, the following format specification is valid (where A is a character array of length at least 8 and size at least 2):

```
A(1) = '(1X,I3,'
A(2) = ' I7, I9)'
PRINT  A, MUTT, AND, JEFF
```

If the expression is an array element, the format must be entirely contained within that element.

If the expression is a character variable, it or any part of it must not be redefined or become undefined during the execution of the I/O statement.

If the expression is a character constant delimited by apostrophes, two apostrophes must be written to represent each apostrophe in the format specification. If a format specification contains, in turn, a character constant delimited by apostrophes, there must be two apostrophes for each of the apostrophe delimiters, and each apostrophe within the character constant must be represented by four apostrophes (see the example below). If quotes are used for the string delimiters and quotes are used within the string, a similar doubling of the quote marks is required. One way to avoid nested delimiter problems is to use delimiters different from the characters within the format specification, if possible. Another way to avoid the problem is to put the character expression in the I/O list instead of the format specification. This is shown in the second line of the following example, where A16 is a character edit descriptor specifying a field width of 16 positions:

```
PRINT '(''I can''''t hear you'')'
PRINT "(A16)", "I can't hear you"
```

The preceding example can be written without a field width (character count), as in the following example:

```
PRINT "(A)", "I can't hear you"
```

When a character expression is used as a format specification, errors in the format specification might not be detected until the program is executed because the format specification may not be complete or known until the data transfer statement is executed.

Format specifications

10.2

Each item in the format item list of a format specification is an *edit descriptor*, which can be a data edit descriptor, control edit descriptor, or character string edit descriptor. Each data list item must have a corresponding data edit descriptor.

Blanks can be used freely in format specifications without affecting the interpretation of the edit descriptors, both in the free and fixed source forms. Named constants are not allowed in format specifications because they would create ambiguities in the interpretation of the format specifications. For example, if N12 were a named integer constant with value 15, the engineering format edit descriptor E N12.4 could be interpreted as the edit descriptor EN12.4 or E15.4.

format_item is defined as follows:

R1003	*format_item*	**is**	[*r*] *data_edit_desc*
		or	*control_edit_desc*
		or	*char_string_edit_desc*
		or	[*r*] (*format_item_list*)
R1004	*r*	**is**	*int_literal_constant*

In the preceding format, *r* must be a default integer literal constant and is called a *repeat factor*. If a repeat factor is not present, it is as if it were present with the value of 1.

r must not have a kind value specified for it. *r* must be a positive integer.

The comma between edit descriptors can be omitted in the following cases:

- Between the scale factor (`P`) and the numeric edit descriptors `F`, `E`, `EN`, `ES`, `D`, or `G`
- Before a new record indicated by a slash when there is no repeat factor present
- After the slash for a new record
- Before or after the colon edit descriptor

Blanks can be used in the following cases:

- Before the first left parenthesis.
- Anywhere in the format specification. Blanks within a character edit descriptor are significant. Blanks outside of a character edit descriptor are not significant.

Edit descriptors can be nested within parentheses, or an edit descriptor can be preceded by a repeat factor indicating that the edit descriptor is repeated. A parenthesized list of edit descriptors can also be preceded by a repeat factor; this indicates that the entire list is to be repeated.

The following examples illustrate many of the edit descriptors that are described in the following subsections.

```
100 FORMAT (2(5E10.1, I10) / (1X, SP, I7, ES15.2))
110 FORMAT (I10, F14.1, EN10.2)
120 FORMAT (TR4, L4, 15X, A20)
130 FORMAT ('MORE SNOW')
140 FORMAT (9X, 3A5, 7/ 10X, 3L4)
```

Data edit descriptor form

10.2.1

Data edit descriptors specify the conversion of values to and from the internal representation to the character representation in the formatted record of a file. The data edit descriptors are defined as follows:

R1005	*data_edit_desc*	**is**	`I` *w* [. *m*]
		or	`B` *w* [. *m*]
		or	`O` *w* [. *m*]
		or	`Z` *w* [. *m*]
		or	`F` *w* . *d*
		or	`E` *w* . *d* [`E` *e*]
		or	`EN` *w* . *d* [`E` *e*]
		or	`ES` *w* . *d* [`E` *e*]
		or	`G` *w* . *d* [`E` *e*]
		or	`L` *w*
		or	`A` [*w*]
		or	`D` *w* . *d*
CRI		**or**	`D` *w* . *d* `E` *e*
CRI		**or**	`R` *w*
R1006	*w*	**is**	*int_literal_constant*
R1007	*m*	**is**	*int_literal_constant*
R1008	*d*	**is**	*int_literal_constant*
R1009	*e*	**is**	*int_literal_constant*

Section 10

ANSI/ISO: The Fortran 90 standard does not specify the `D`*w* `.` *d* `E` *e* or `R`*w* data edit descriptors.

The *w*, *m*, *d*, and *e* descriptors must be default integer literal constants. They have the following meanings:

Descriptor	Specification
w	The width of the field
m	The least number of digits in the field
d	The number of fractional digits in the field
e	The number of digits in the exponent

w, *m*, *d*, and *e* must not have a kind value specified for them.

w and *e* must be positive.

The `I`, `B`, `O`, `Z`, `F`, `E`, `EN`, `ES`, `G`, `L`, `A`, and `D` edit descriptors indicate the manner of editing.

The meanings of the data edit descriptors are described in subsection 10.5, page 473, through subsection 10.7, page 487.

Control edit descriptor form

10.2.2

Control edit descriptors determine the position, form, layout, and interpretation of characters transferred to and from formatted records in a file. The control edit descriptors are defined as follows:

R1010	*control_edit_desc*	**is**	*position_edit_desc*
		or	[*r*] `/`
		or	**:**
		or	*sign_edit_desc*
		or	*k* `P`
		or	*blank_interp_edit_desc*
R101 1	*k*	**is**	*signed_int_literal_constant*
R1012	*position_edit_desc*	**is**	`T` *n*
		or	`TL` *n*
		or	`TR` *n*
		or	*n* `X`
CRI		**or**	`$`
R1013	*n*	**is**	*int_literal_constant*
R1014	*sign_edit_desc*	**is**	`S`
		or	`SP`
		or	`SS`
R1015	*blank_interp_edit_desc*	**is**	`BN`
		or	`BZ`

Section 10

n and *r* must be default integer literal constants. *k* must be a signed default integer literal constant. *n*, *k*, and *r* must not have a kind value specified for them. *k*, *n*, and *r* have the following meanings:

Descriptor	Specification
k	A scale factor.
n	A position in the record to move, relative to the left tab limit, for descriptor `T`. Specifies the number of spaces to move for descriptors `X`, `TR`, and `TL`. *n* must be positive.
r	The repeat factor.

The control edit descriptors `T`, `TL`, `TR`, `X`, and `$` are called *position edit descriptors* (R1012). The control edit descriptors `S`, `SP`, and `SS` are called *sign edit descriptors* (R1014). The control edit descriptors `BN` and `BZ` are called *blank interpretation edit descriptors* (R1015).

> **ANSI/ISO:** The Fortran 90 standard does not describe the `$` as a position edit descriptor.

In *k*`P`, *k* is called the *scale factor.*

`T`, `TL`, `TR`, `X`, `$`, slash, colon, `S`, `SP`, `SS`, `P`, `BN`, and `BZ` indicate the manner of editing and are described in detail in subsection 10.8, page 489.

Character string edit descriptor form

10.3

Character string edit descriptors specify character strings to be transmitted to the formatted output record of a file. The character string edit descriptors are defined as follows:

R1016	*char_string_edit_desc*	**is**	*char_literal_constant*
		is	*c* `H` *rep_char* [*rep_char*] ...
R1017	*c*	**or**	*int_literal_constant*

c must be a default integer literal constant and is a character count. *c* must not have a kind value specified for it. *c* must be positive.

Section 10

The second form of the character edit descriptor is obsolescent.

The character string edit descriptors are described in detail in subsection 10.10, page 497.

Formatted data transfer

10.4

The format specification indicates how data are transferred by `READ`, `WRITE`, and `PRINT` statements. The data transfer typically involves a conversion of a data value. The particular conversion depends on the next data input or output item, along with the current edit descriptor in the format specification.

```
        READ (*, '(A7, I10, E16.8)' )  X, Y, Z
        WRITE (*, 100)  X, Y, Z
    100 FORMAT (A7, I10, E16.3)
```

An empty format specification ( ) is restricted to I/O statements with no items in the I/O data item list or a list of items all of which have zero size. A zero-length character string requires an `A` edit descriptor.

The effect on I/O of an empty format specification depends on whether the data transfer is advancing or nonadvancing, and on whether there is a current record. The effect is described by the following cases.

When the data transfer is advancing:

- If there is no current record, then:
  - On input, skip the next record
  - On output, write an empty record
- If there is a current record, then:
  - On input, skip to the end of the current record
  - On output, terminate the current record

When the data transfer is nonadvancing:

- If there is no current record, then:
  - On input, move to the initial point of the next record
  - On output, create an empty record and move to its initial point
- If there is a current record, then:
  - On input, there is no effect
  - On output, there is no effect

The following example program segment reads N character strings, each of length M, from a single record and then advances to the beginning of the next record:

```
DO I = 1, N
   READ (5, '(A1)', ADVANCE='NO') &
       (CHARS(I)(J:J), J = 1, M)
ENDDO
READ (5, '()', ADVANCE = 'YES')
```

The data and the edit descriptors are converted in a left-to-right fashion, except for repeated items, which are repeated until either the data items are exhausted or the repeat number is reached. A complex data item requires two data edit descriptors for data items of type real; that is, two of the edit descriptors E, F, D, ES, EN, or G. The two edit descriptors may be different.

Control edit descriptors and character edit descriptors do not require a corresponding data item in the list. The effect is directly on the record transferred. When the data items are completed, no further change is made to record on output, and no change is made to the position in the file on input.

Parentheses usage

10.4.1

The effect of parentheses in a format specification depends on the nesting level of the parentheses.

When the rightmost right parenthesis of a complete format specification is encountered and there are no more data items, the I/O data transfer terminates. Remember that the format specification can be given by a character string expression. In such a case, the right parenthesis matching the leftmost left parenthesis can be followed by any characters, including parentheses. None of these trailing characters are relevant to the rules and restrictions in this section. For example, the following character string can be used as a format specification in a character string, and the part after the first right parenthesis is ignored:

```
'(I5,E16.8,A5)  (This part is ignored)'
```

If there are more data items when the rightmost right parenthesis is encountered, format control continues beginning at the left parenthesis corresponding to the last preceding right parenthesis in the specification, if there is one, with an implied slash (/) to cause a new record to begin. If there is no preceding right parenthesis, the reversion is to the beginning of the format.

If there is a repeat factor encountered when reverting, the repeat before the parenthesis is reused.

Reversion does not affect the scale factors, the sign control edit descriptor, or blank interpretation. These remain in effect for the duration of the format action.

```
CHR_FMT = '(I5, 4(3F10.2, 10X), E20.4)'
```

In the previous example, if the character string were used in a formatted output data transfer statement, the first output data item must be an integer . The remaining items must be of type real (or complex); 13 real values are written on the first record after the integer, and the next real values are written in each new record, 13 at a time, until the data items are exhausted. All but the last record will have 13 real values written, 3 real values using the `F10.2` edit descriptor, 10 blanks, followed by 3 more real values and 10 blanks repeated 4 times in total, followed by a real value using the `E20.4` edit descriptor. This behavior is described in more detail in the next subsection.

Correspondence between a data-edit descriptor and a list item

10.4.2

The best way to describe how this correspondence is determined is to think of two markers, one beginning at the first item of the I/O data item list and the other beginning at the first left parenthesis of the format specification. Before describing how each marker proceeds through each list, the I/O data item list is considered to be expanded by writing out each element of an array , each component of a structure, each part (real and imaginary) of each item of type complex, and each iteration of each implied-DO list. The expanded item list is called the *effective data item list,* and each item in the list is called an *effective item.* Note that zero-sized arrays yield no effective items, but zero-length character objects yield effective items. Also, the format specification is considered expanded for each repeat factor preceding any data or slash edit descriptor but not a parenthesized format item list. If the data item list is nonempty, there must be at least one data edit descriptor in the format specification. Given the effective data item list and expanded format specification, the markers proceed as follows:

1. The marker proceeds through the format specification until the first data edit descriptor or right parenthesis is encountered. Any control edit descriptor or string edit descriptor encountered before the first data edit descriptor is encountered is interpreted according to its definition, each possibly changing the position within the record or the position within the file, or changing the interpretation of data in the record or conversion of data to the record.

2. If a data edit descriptor is encountered first, the effective data item pointed to by the marker in the data item list is transferred and converted according to the data edit descriptor, and the marker in the data item list proceeds to the next effective data item.

3. If a right parenthesis is encountered and the right parenthesis is not the outermost one of the format specification, the repeat factor in front of the matching left parenthesis is reduced by one. If the reduced factor is nonzero, the marker scans right from this left parenthesis, looking for a data edit descriptor as above. If the repeat factor becomes zero, the format specification marker then proceeds right from the right parenthesis, again looking for a data edit descriptor. If the right parenthesis is the outermost right parenthesis, the marker reverts to the left parenthesis corresponding to the last preceding right parenthesis, if

there is one; if there is no preceding right parenthesis, it reverts to the first left parenthesis of the format specification. Upon reversion, a slash edit descriptor is interpreted implicitly, and the format marker proceeds right from this left parenthesis, honoring any repeat factor in front of it.

4. If no effective data item remains when a data edit descriptor is encountered or when a colon edit descriptor is encountered, the I/O operation terminates.

To illustrate how this works, consider the following example:

```
INTEGER  A(3)
COMPLEX  C
TYPE RATIONAL
   INTEGER  N, D
END TYPE
TYPE(RATIONAL)  R
   . . .
WRITE (*, &
     "('A and C appear on line 1, R appears on line 2' &
      / (1X, 3I5, 2F5.2) )" )  A, C, R
```

The data item list is first expanded as described above. The expanded data item list becomes the following:

```
     A(1), A(2), A(3), REAL(C), AIMAG(C), R%N, R%D
```

The format specification is also expanded and becomes the following:

```
     ('A and C appear on line 1, R appears on line 2' &
        / (1X, I5, I5, I5, F5.2, F5.2) )
```

A marker is established in the data item list, which initially points at the item `A(1)`. A marker is also established in the format specification and initially points to the first left parenthesis. The marker in the format specification proceeds right to the first edit descriptor, which is the first `I5`. In so doing, it sees the string edit descriptor (which is transferred to the output record), the slash edit descriptor (which causes the previous record to terminate and to begin a new record), and the position edit descriptor (which positions the record at the second character, blank filling the first character). The item `A(1)` is then converted according to the `I5` specification and the converted value is transferred to the output record. The marker in the data item list is moved to `A(2)`. The format specification marker is moved right to the second I5 edit descriptor, and A(2) is converted and transferred to the output record. Similarly, A(3), the real part of C, and the imaginary part of C are converted and transferred to the output record.

At this point, the data item list marker is pointing at R%N, and the format specification marker begins scanning after the second F5.2 edit descriptor looking for the next edit descriptor. The first right parenthesis is encountered and the scan reverts back to the corresponding left parenthesis. The repeat factor in front of this parenthesis is 1 by default and is reduced by 1 to 0. The marker in the format specification proceeds right from the first right parenthesis, encountering the outermost right parenthesis and then reverts to the left parenthesis before the edit descriptor 1X. As a result, an implicit slash edit descriptor is interpreted, causing the previous output record to be completed and a new record to be started. The format specification marker scans right looking for a data edit descriptor, which is the first I5. In the process of the scan right, the position edit descriptor is interpreted, which positions the file at the second character of the next record (and blank fills the skipped character).

Finally , the N and D components of R are converted and transferred to the output record, using the first two I5 edit descriptors. The data item list marker finds no further items, so the output operation terminates.

The following is an example of writing a zero-sized array and zero-length character string using formatted output data transfer:

```
REAL  A(10)
CHARACTER(4)  CHR
    . . .
WRITE(6, '()')  A(1:0)
WRITE(6, '(A4)')  CHR(4:3)
```

An empty format specification is allowed for the first WRITE statement, because the array to be printed is a zero-sized array section. The format specification in the second WRITE statement is required to have at least one A edit descriptor because the effective data item is a zero-length character string, not a zero-sized array. In the first case, an empty record is written, and in the second case, a record consisting of four blank characters is written.

Section 10

File positioning by format control

10.5

Assume that there is a current record being processed. After each data edit descriptor is used, the file position within that record is following the last character read or written by the particular edit descriptor. On output, after a string edit descriptor is used, the file is positioned within that record following the last character written. (See the description of the control edit descriptors `T`, `TL`, `TR`, `X`, and `$` for any special positioning within the current record; see the description of the slash edit descriptor for special positioning within the file.) The remaining control edit descriptors do not affect the position within a record or within the file; they affect only the interpretation of the input characters or the form of the output character string or how subsequent edit descriptors are interpreted. The interpretation of the edit descriptors is not affected by whether the operation is an advancing or nonadvancing I/O operation.

Numeric editing

10.6

There are seven edit descriptors that cover numeric editing: `I`, `F`, `E`, `EN`, `ES`, `D`, and `G`. The following rules apply to all of them:

On input:

- Leading blanks are never significant.
- Plus signs can be omitted in the input data.
- A blank field is considered to be zero, regardless of the `BN` edit descriptor or the `BLANK=` specifier in effect.
- Within a field, blanks are interpreted in a manner that depends on the `BLANK=` specifier default for preconnected files, the `BLANK=` specifier provided in an `OPEN` statement for the unit, or if there is a `BN` or `BZ` blank edit descriptor in effect.
- In numeric fields that have a decimal point and correspond to `F`, `E`, `EN`, `ES`, `D`, or `G` edit descriptors, the decimal point in the input field overrides the placement of the decimal point specified by the edit descriptor specification.

Section 10

- Data input is permitted to have more digits of significance than can be used to represent a number.
- The lowercase exponent letters `e` and `d` are equivalent to the corresponding uppercase exponent letters.

On output:

- A positive or zero value may have a plus sign, depending on the sign edit descriptors used.
- Negative values have a negative sign unless the printed value would be zero (because of the conversion rules). Negative zero is never produced.

  For example, suppose variable `SMALL` has the value `-0.000314`. On output, the characters transferred to output unit 6 by the following statements will not contain a negative sign:

  ```
       WRITE(6,10)  SMALL
  10   FORMAT(F5.2)
  ```

  The field will be as follows because a negative zero cannot be printed:

  b`0.00`

- The number is right justified in the field. Leading blanks may be inserted.
- If the number or the exponent is too large for the field width specified in the edit descriptor, the entire output field is filled with asterisks.
- Asterisks are not produced when the optional characters can be omitted and the output character string fits in the output field.

Integer editing
10.6.1

The Fortran 90 standard states that the I/O list item corresponding to an integer edit descriptor must be of type integer, except for the `G` edit descriptor . The integer edit descriptors are as follows:

`I`*w* [`.`*m*]

Section 10

```
Bw [ .m ]
```

```
Ow [ .m ]
```

```
Zw [ .m ]
```

```
Gw.d [ Ee ]
```

w, *m*, *d*, and *e* have the following meanings:

Descriptor	Specification
w	The width of the field
m	The minimum number of digits in the field
d	The number of fractional digits in the field
e	The number of digits in the exponent

For both input and output:

- The value of *m* must not exceed the value of *w*.
- For an integer I/O list item, the edit descriptor G*w*.*d*[E*e*] follows the same rules as the I*w* edit descriptor for the given value of *w*.

On input:

- *m* has no effect on an input field.
- For the I edit descriptor, the character string in the file must be an optionally signed integer constant, but it cannot have an underscore or a kind parameter value.

- For the B, O, or Z edit descriptors, the character string must be a string of blanks, a sign, and digits of binary, octal, or hexadecimal base, respectively . For example, the character string corresponding to a B edit descriptor must not contain digits 2 through 9. The character string corresponding to an O edit descriptor must not contain the digits 8 or 9. The character string corresponding to a Z edit descriptor may consist of the digits 0 through 9, the letters A through F, or the letters a through f.

  ANSI/ISO: The Fortran 90 standard allows only binary, octal, or hexadecimal digits and blanks as input.

On output:

- If *m* is not present, each edit descriptor behaves as if it were present with the value 1.
- For the I*w.m* edit descriptor with *m* positive, the field is *w* characters wide and consists of zero or more leading blanks, followed by an optional sign, followed by an unsigned integer consisting of at least *m* digits. Leading zeros pad an integer field until there are *m* digits.
- For the I*w*.0 edit descriptor, if the output list item has the value 0, the field consists entirely of blanks.
- If *m* is not present and the value of the output list item is nonzero, the first digit must be nonzero; otherwise, the field consists of only one 0 digit with no sign character. Negative zero is never produced.
- For the B, O, or Z edit descriptors, the rules for forming the output field for the values *w* and *m* are the same as for the I edit descriptor except that the signed integer must consist of digits from the binary, octal, or hexadecimal base, respectively. A negative value is indicated in the encoding of the digits written; that is, a minus sign is not written.

  ANSI/ISO: The Fortran 90 standard allows only an unsigned integer as an output list item.

Real editing
10.6.2

The F, E, EN, ES, and D edit descriptors specify editing for real and complex I/O list items. The G edit descriptor may also be used for real and complex items. Two such edit descriptors are required for each complex data item.

The forms of the edit descriptors for real values are as follows:

```
Fw.d
```

```
Ew.d [ Ee ]
```

```
ENw.d [ Ee ]
```

```
ESw.d [ Ee ]
```

```
Dw.d
```

```
Gw.d [ Ee]
```

```
Dw.d Ee
```

w, *d*, and *e* have the following meanings:

Descriptor	Specification
w	The width of the field
d	The number of fractional digits in the field
e	The number of digits in the exponent

`F` editing
10.6.2.1

`F`*w*.*d* editing converts to or from a string occupying *w* positions.

For both input and output:

- *d* must not exceed *w*.
- The value in the input field or the value transferred to the output field may be signed.

On input:

- *d* specifies the number of decimal places in the input value if a decimal point is not present in the input field.
- The input field may be one of the following:
  - A signed integer or real literal constant but without an underscore and kind parameter value
  - A signed digit string followed by a sign followed by an unsigned digit string treated as an exponent
  - A signed digit string containing a decimal point followed by a sign followed by an unsigned digit string treated as an exponent

  Blanks may be freely inserted anywhere in the input field.
- If the input field contains a decimal point, the value of *d* has no effect.
- If there is no decimal point, a decimal point is inserted in front of the rightmost *d* digits of the nonexponent part, treating blanks as 0 digits or as if they were not present, according to the `BLANK=` specifier or the `BZ` or `BN` edit descriptor currently in effect.

  Consider the format specification `F5.1`. The following input data item is treated as the real number `19.9`, if the `BLANK=` specifier is `NULL` or the `BN` edit descriptor is in effect, and is treated as the real number `1009.9` if the `BLANK=` specifier is `ZERO` or the `BZ` edit descriptor is in effect:

  `1`*bb*`99`
- There might be more digits in the number than the internal representation can contain.

Section 10

- The number can contain an `E` or `D` indicating an exponent value; a field with a `D` exponent letter is processed identically to the same field with an `E` exponent letter. If there is no exponent field on input, the assumption is that the character string is followed by an exponent with the value $-k$ where k is the scale factor established by a previous *k*`P` edit descriptor.

On output:

- *d* specifies the number of digits after the decimal point.
- The form of the output field consists of *w* positions comprised of leading blanks, if necessary, and an optionally signed real constant with a decimal point, rounded to *d* digits after the decimal point but with no exponent, underscore, or kind parameter value.
- Leading zeros are not written unless the number is less than 1.
- At least one zero is written if no other digits would appear.
- The scale factor has no effect on `F` editing on output.
- Negative zero is never produced.

Example:

```
      READ (5, 100)  X, Y
  100 FORMAT (F10.2, F10.3)
```

Assume the following input field:

bbbb `6.42181234567890`

The values assigned to `X` and `Y` are `6.4218` and `1234567.89`, respectively. The value of *d* is ignored for `X` because the input field contains a decimal point.

`E` and `D` editing 10.6.2.2

The `E`*w*`.`*d*[`E`*e*], `D`*w*`.`*d*`E`*e*, and `D`*w*.*d* edit descriptors convert to and from a string occupying *w* positions. For the edit descriptors `E`*w*`.`*d*[`E`*e*], `D`*w*`.`*d*`E`*e*, and `D`*w*.*d*, the field representing the floating point number contains *w* characters, including an exponent.

Section 10

For both input and output:

- w is the field width, d is the number of places after the decimal, and e is the exponent width.
- d and e must not exceed w.

On input:

- The forms `E`*w*`.`*d*[`E`*e*], `D`*w*`.`*d*`E`*e*, and `D`*w*.*d* are the same as for `F`*w*`.`*d* editing, where either `E` or `D` in the input data record may indicate an exponent. e has no effect on input.

On output:

- The form of the output field for a scale factor of zero is as follows:

  [$\pm$] [0] . x_1x_2 ... x_d *exp*

  $\pm$ signifies a plus or a minus.

  x_1x_2 ... x_d are the d most significant digits of the data value after rounding.

  exp is a decimal exponent having one of the forms specified in Table 33, where each z_i is a decimal digit.

Table 33. Forms for the exponent *exp* in `E` and `D` editing

Edit descriptor	Absolute value of exponent	Form of exponent
`E`*w*`.`*d*	$\lvert exp \rvert \le 99$	$E\pm z_1z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1z_2z_3$
	$999 < \lvert exp \rvert \le 2466$	$\pm z_1z_2z_3z_4$
`E`*w*`.`*d*`E`*e*	$\lvert exp \rvert \le 10^e - 1$	$E\pm z_1z_2 \ldots z_e$

Section 10

Table 33. Forms for the exponent *exp* in E and D editing (continued)

Edit descriptor	Absolute value of exponent	Form of exponent
D*w*.*d*	$\lvert exp \rvert \le 99$	E$\pm z_1 z_2$
	$99 < \lvert exp \rvert \le 999$	$\pm z_1 z_2 z_3$
	$999 < \lvert exp \rvert \le 2466$	$\pm z_1 z_2 z_3 z_4$
D*w*.*d*E*e*	$\lvert exp \rvert \le 10^e - 1$	E$\pm z_1 z_2 \ldots z_e$

ANSI/ISO: The Fortran 90 standard restricts the use of E*w*.*d* and D*w*.*d* to an exponent less than or equal to 999. The E*w*.*d*E*e* form must be used.

- The sign in the exponent is always written.
- Plus is used for zero exponents.
- A scale factor *k*P can be used to specify the number of digits to the left of the decimal point, with the exponent adjusted accordingly; that is, the scale factor *k* controls the decimal normalization. If $-d < k \le 0$, the output field contains the decimal point, exactly $\lvert k \rvert$ leading zeros, and $d - \lvert k \rvert$ significant digits. If $0 < k < d + 2$, the output field contains exactly *k* significant digits to the left of the decimal point and $d - k + 1$ significant digits to the right of the decimal point. Other values of *k* are not permitted; that is, those values of *k* that will produce no digits to the left of the decimal point or specify fewer than zero digits to the right of the decimal point. The output field will contain *w* asterisks.
- The form of zero on output always contains a decimal point, *d* zero digits, and an exponent of 4 characters whose digits are zero.

Consider the following example:

```
      WRITE (6, 105)  Y, Z
  105 FORMAT (E15.3,4PD15.3)
```

Section 10

If the values of `Y` and `Z` are `-21.2` and `26542.1232` respectively, the output record produced is as follows:

bbbbb `-0.212E+02`*bbb* `2654.212E+01`

Engineering edit descriptor `EN`
10.6.2.3

The `EN` edit descriptor converts to or from a string using engineering notation for a value occupying *w* positions.

On input:

- The form `EN`*w*`.`*d*[`E`*e*] is the same as for `F`*w*`.`*d* editing.

On output:

- The output of the number is in the form of engineering notation, where the exponent is divisible by 3 and the absolute value of the significand is `1000 >` |*significand* | ≥ `1`. This is the same form as the `E` edit descriptor, except for these exponent and significand differences.

```
      WRITE (6, 110)  B
  110 FORMAT (EN13.3)
```

  If the value of `B` is `0.212`, the output record produced is as follows:

  bb`212.000E-03`

- The form of zero is the same as noted for the `E` edit descriptor except that there is exactly one zero digit before the decimal point.

- The form of the output field is as follows:

  [±] *yyy* . x_1x_2 ... x_d *exp*

  ± signifies a plus or a minus.

  yyy are the 1 to 3 decimal digits representing the most significant digits of the value of the data after rounding (*yyy* is an integer such that `1` ≤ *yyy* < `1000` or, if the output value is zero, *yyy* = `0`).

  x_1x_2 ... x_d *exp* are the *d* next most significant digits of the value of the data after rounding. If the output value is zero, the x_i are all 0.

exp is a decimal integer, divisible by 3, representing the exponent and of one of the forms, given in Table 34, where each z_i is a decimal digit.

Table 34. Forms for the exponent *exp* in `EN` editing

Edit descriptor	Absolute value of exponent	Form of exponent
EN*w*.*d*	$\lvert exp \rvert \leq 99$	$\mathrm{E}\pm z_1 z_2$
	$99 < \lvert exp \rvert \leq 999$	$\pm z_1 z_2 z_3$
	$999 < \lvert exp \rvert \leq 2466$	$\pm z_1 z_2 z_3 z_4$
EN*w*.*d*E*e*	$\lvert exp \rvert \leq 10^e - 1$	$\mathrm{E}\pm z_1 z_2 \ldots z_e$

- The sign in the exponent is always written. A plus sign is written if the exponent value is zero. The form EN*w*.*d*E*e* must be used with a sufficiently large value of *e* if $\lvert exp \rvert > 999$.

Internal value	Output field using `SS, EN12.3`
6.421	6.421E+00
-.5	-500.000E-03
.00217	2.170E-03
4721.3	4.721E+03

10.6.2.4 Scientific edit descriptor `ES`

The scientific edit descriptor `ES` converts to or from a string using scientific notation for a value occupying *w* positions.

On input:

- The form ES*w*.*d*[E*e*] is the same as for F*w*.*d* editing.

Section 10

On output:

- The output of the number is in the form of scientific notation, where the absolute value of the significand is $10 > |significand| \geq 1$. This is the same form as the `E` edit descriptor, except for the significand difference.

```
     WRITE (6, 110)  B
110 FORMAT (ES12.3)
```

  If the value of `B` is `0.12345678`, the output record produced is as follows:

  bbb`1.235E-01`

- The form of zero is the same as noted for the `EN` edit descriptor.

- The form of the output field is as follows:

  $[\pm]\ y.x_1x_2 \ldots x_d\ exp$

  $\pm$ signifies a plus or a minus.

  y is a decimal digit representing the most significant digit of the value of the data after rounding (*y* is an integer such that $1 \leq y < 10$ or, if the output value is zero, $y = 0$).

  $x_1x_2 \ldots x_d$ are the *d* next most significant decimal digits of the value of the data after rounding. If the output value is zero, the x_i are all 0.

  exp is a decimal exponent of one of the forms, given in Table 35, where each z_i is a decimal digit.

Table 35. Forms for the exponent *exp* in `ES` editing

Edit descriptor	Absolute value of exponent	Form of exponent
`ES`*w*`.`*d*	$\lvert exp \rvert \leq 99$	$E\pm z_1z_2$
	$99 < \lvert exp \rvert \leq 999$	$\pm z_1z_2z_3$
	$999 < \lvert exp \rvert \leq 2466$	$\pm z_1z_2z_3z_4$
`ES`*w*`.`*d*`E`*e*	$\lvert exp \rvert \leq 10^e - 1$	$E\pm z_1z_2 \ldots z_e$

Section 10

- The sign in the exponent is always written.
- A plus sign is written if the exponent value is zero.
- The form `ES`*w*`.`*d*`E`*e* must be used with a sufficiently large value of *e* if |*exp*| `> 999`.

Internal value	Output field using `SS`, `ES12.3`
6.421	6.421E+00
-.5	-5.000E-01
.00217	2.170E-03
4721.3	4.721E+03

Complex editing
10.6.2.5

Complex editing follows the rules for numeric editing. Editing of complex numbers requires two real edit descriptors, the first one for the real part and the second one for the imaginary part. Different edit descriptors may be used for the two parts. Control and character string edit descriptors can be inserted between the edit descriptors for the real and imaginary parts.

```
COMPLEX CM(2)
READ(5, "(4E7.2)") (CM(I), I = 1, 2)
```

Assume that the input record is as follows:

bb`55511`*bbb*`2146`*bbbb*`100`*bbbb*`621`

The values assigned to `CM(1)` and `CM(2)` are `555.11 + 21.46`*i* and `1.0 + 6.21`*i*, respectively.

Generalized editing of real data
10.6.2.6

`G`*w*`.`*d*[`E`*e*] converts to or from a string using generalized editing. The form for generalized editing is determined by the magnitude of the value of the number .

On input:

- The `G`*w*`.`*d*[`E`*e*] edit descriptor is the same as the `F`*w*`.`*d* edit descriptor.

On output:

- Assume that N is the magnitude of a number to be printed using a `G` edit descriptor . If $N = 0$ and d is not zero, or if N is approximately between `0.1` and `10`^{d}, Table 36 specifies the form of the output, where n is `4` for `G`w`.`d and n is $e + 2$ for `G`w`.`d`E`e. A k`P` scale factor has no effect.

- If N is outside this range, output editing with the edit descriptor k`PG`w`.`d[`E`e] is the same as that with k`PE`w`.`d[`E`e].

Table 36. The form of the output using a `G` edit descriptor for a number of magnitude N

Magnitude of data	s	t	Equivalent conversion
$N = 0$	$w - n$	$d - 1$	`F`s`.`t`,` n`X`
$0.1 - 0.5 \times 10^{-d-1} \leq N < 1 - 0.5 \times 10^{-d}$	$w - n$	d	`F`s`.`t`,` n`X`
$1 - 0.5 \times 10^{-d} \leq N < 10 - 0.5 \times 10^{-d+1}$	$w - n$	$d - 1$	`F`s`.`t`,` n`X`
$10 - 0.5 \times 10^{-d+1} \leq N < 100 - 0.5 \times 10^{-d+2}$	$w - n$	$d - 2$	`F`s`.`t`,` n`X`
.	.		
.	.		
.	.		
$10^{d-2} - 0.5 \times 10^{-2} \leq N < 10^{d-1} - 0.5 \times 10^{-1}$	$w - n$	1	`F`s`.`t`,` n`X`
$10^{d-1} - 0.5 \times 10^{-1} \leq N < 10^{d} - 0.5$	$w - n$	0	`F`s`.`t`,` n`X`

Key:
`X` indicates blanks.
n is 4 for `G`w`.`d edit descriptors.
n is e+2 for `G`w`.`d`E`e edit descriptors.

Example 1:

```
PRINT "(G10.1)", 8.76E1
```

The preceding statement produces the following output:

bbb `0.9E+02`

The magnitude of *N* is outside the range for an `F` edit descriptor; it yields the format `E10.1`.

Example 2:

```
PRINT "(G10.3)", 8.76E1
```

The preceding statement produces the following output:

bb `87.6`*bbbb*

With the `G10.3` edit descriptor, *n* is 4, and the format reduces to `F6.1,4X` (the fourth line in Table 36) because of the magnitude of the number.

Example 3:

```
PRINT "(G10.3E1)", 8.76E1
```

The preceding statement produces the following output because *n* is `3 (=1+2)`, and the format reduces to `F7.1,3X`:

bbb `87.6`*bbbb*

Logical editing

10.7

The logical edit descriptors convert to or from a string representing a logical value that is true or false. The edit descriptors used for logical editing are as follows:

`L`*w*

```
Gw.d [ Ee ]
```

w, *d*, and *e* have the following meanings:

Descriptor	Specification
w	The width of the field
d	The number of fractional digits in the field
e	The number of digits in the exponent

For both input and output:

- Generalized logical editing `G`*w*`.`*d*[`E`*e*] follows the rules for `L`*w* editing.

On input:

- The input field for a logical value consists of any number of blanks, followed by an optional period, followed by `T` or `F`, for a true or false value respectively , followed by any representable characters.
- The `T` or `F` and any following letters can be in lowercase. They are treated the same as the uppercase letters.

  For example, if you use the following `READ` statement:

  ```
  READ(5, "(2L8)") L1, L2
  ```

  to read the following input record:

  ```
  .TRUE.bb.Falseb
  ```

  `L1` and `L2` will have the values true and false, respectively. The result would be the same if the input record were as follows:

  ```
  TUESDAYbFRIDAYbb
  ```

On output:

- The output field consists of *w* - 1 leading blanks, followed by `T` or `F`, for a true or false value, respectively.

  ```
  WRITE(6, "(2L7)") L1, L2
  ```

  If `L1` and `L2` are true and false, respectively, the output record will be as follows:

  bbbbbb `T`*bbbbbb* `F`

Character editing

10.8

Character editing converts to or from a string of characters. The edit descriptors for character editing are as follows:

```
A [ w ]
```

```
Gw.d [ Ee ]
```

w, *d*, and *e* have the following meanings:

Descriptor	Specification
w	The width of the field
d	The number of fractional digits in the field. *d* is ignored when `G` is used for character editing.
e	The number of digits in the exponent. *e* is ignored when `G` is used for character editing.

For both input and output:

- *w* is the field width measured in characters.
- A `G`*w*`.`*d*[`E`*e*] general edit descriptor is the same as an `A`*w* edit descriptor for character data.

Section 10

- If *w* is omitted from the A format, the length of the character data object being transferred is used as the field width.

On input:

- If *w* is greater than or equal to the length *len* of the character data read, *len* rightmost characters of the input field are read.
- If *w* is less than the length *len* of the character data read, the *w* characters of the character data will be read from the input field and placed left justified in the character list item followed by *len* − *w* trailing blanks.

On output:

- If *w* exceeds the length *len* of the character data written, *w* − *len* blank padding characters are written followed by *len* characters of the character data.
- If *w* is less than or equal to the length *len* of the character data written, the *w* leftmost characters of the character data will appear in the output field.

```
CHARACTER(LEN = 14), PARAMETER :: &
      SLOGAN = "SAVE THE RIVER"
WRITE (*, "(A)")  SLOGAN
```

The preceding lines produce the following output record:

```
SAVE THE RIVER
```

Control edit descriptors

10.9

No data is transferred or converted with the control edit descriptors. Control edit descriptors affect skipping, tabbing, scale factors, and printing of optional signs. These edit descriptors may affect how the data is input or output using the subsequent data edit descriptors in the format specification.

Position editing
10.9.1

Position edit descriptors control relative tabbing left or right in the record before the next list item is processed. The edit descriptors for tabbing, where *n* is a positive integer, are as follows:

Descriptor	Meaning
`T` *n*	Tab to position *n*
`TL` *n*	Tab left *n* positions
`TR` *n*	Tab right *n* positions
n `X`	Tab right *n* positions
`$`	Carriage control

> **ANSI/ISO:** The Fortran 90 standard does not specify the dollar sign (`$`) edit descriptor.

The tabbing operations to the left are limited by a position called the *left tabbing limit*. This position is normally the first position of the current record but, if the previous operation on the file was a nonadvancing formatted data transfer, the left tabbing limit is the current position within the record before the data transfer begins. If the file is positioned to another record during the data transfer, the left tabbing limit changes to the first position of the new record.

The `T`*n* edit descriptor positions the record just before the character in position *n* relative to the left tabbing limit. `TR`*n* and *n*`X` move right *n* characters from the current position. `TL`*n* moves left *n* characters from the current position, but is limited by the left tabbing limit.

For both input and output:

- *n* must be a positive integer constant with no kind parameter value specified for it.
- Left tabbing is always limited so that even if left tabbing specifies a position to the left of the left tabbing limit, the record position is set to the left tabbing limit in the record.

Section 10

- The left tabbing limit in the record is determined by the position in the record before any data transfer begins for a particular data transfer statement.
- If a file is positioned to another record during a particular data transfer statement, the left tabbing limit is the first position of the record.

On input:

- The `T` descriptor might move the position within a record either left or right from the current position.
- Moving to a position left of the current position allows input to be processed twice, provided the same input statement is performing the processing for advancing input, or provided nonadvancing input is in effect.
- The `X` descriptor always moves the position to the right and skips characters.

On output:

- The positioning does not transmit characters, and does not by itself cause the record to be shorter or longer.
- Positions that are skipped and have not been filled previously behave as if they are blank filled.
- Positions previously filled can be replaced with new characters, but are not blank filled when they are skipped using any of the position edit descriptors.

For example, assume that `DISTANCE` and `VELOCITY` have the values `12.66` and `-8654.123`.

```
      PRINT 100, DISTANCE, VELOCITY
  100 FORMAT (F9.2, 6X, F9.3)
```

The preceding code produces the following record:

bbbb `12.66`*bbbbbb* `-8654.123`

Section 10

Continue to assume that DISTANCE and VELOCITY have the values 12.66 and -8654.123.

```
      PRINT 100, DISTANCE, VELOCITY
  100 FORMAT(F9.2, T7, F9.3)
```

The preceding code produces the following record because T7 specifies the first position for VELOCITY as the seventh character in the record:

bbbb 12-8654.123

The dollar sign character ($) in a format specification modifies the carriage control specified by the first character of the record. In an output statement, the $ descriptor suppresses the carriage return or line feed. In an input statement, the $ descriptor is ignored. It is intended primarily for interactive I/O.

Slash editing
10.9.2

The slash edit descriptor consists of the single slash character (/). The current record is ended when a slash is encountered in a format specification.

On input:

- If the file is connected for sequential access, the file is positioned at the beginning of the next record. The effect is to skip the remainder of the current record.
- For direct access, the record number is increased by one, and the file is positioned at the beginning of the record with this increased record number , if it exists; it becomes the current record.
- A record can be skipped entirely on input.

On output:

- If the file is connected for sequential access, the current record is written. The current record can be an empty record.
- For direct access, the current record is blank filled, if necessary, the record number is increased by one, and this record becomes the current record.

Section 10

- For an internal file that is a scalar, the current record is blank-filled, if necessary . No other action is taken until a check of the contents of the format is done; that is, the format determines what happens next.
- For an internal file that is an array, the current record is blank filled, and the file is positioned at the beginning of the next array element.

Assume in the following code that `ALTER`, `POSITION`, and `CHANGE` have the values `1.1`, `2.2`, and `3.3`, respectively:

```
PRINT "(F5.1, /, 2F6.1)", ALTER, POSITION, CHANGE
```

The preceding code produces the following two records:

```
bb1.1
bbb2.2bbb3.3
```

Colon editing

10.9.3

The colon edit descriptor consists of the character colon (`:`). If the list of items in a formatted `READ` or `WRITE` statement is exhausted, a colon stops format processing at that point. If the list is not exhausted, the colon edit descriptor has no effect.

For example, assume in the following code that `ALTER`, `POSITION`, and `CHANGE` have the values `1.1`, `2.2`, and `3.3`, respectively:

```
WRITE(6, 100)  ALTER, POSITION, CHANGE
100 FORMAT(3F5.2, :, "STOP")
```

The preceding code produces the following:

```
bb1.1bb2.2bb3.3
```

The characters `STOP` are not printed because the output list is exhausted when the colon edit descriptor is processed. If the colon edit descriptor were not present in the above format, the string `"STOP"` would be printed.

Sign editing

10.9.4

Sign editing applies to the output data transfer of positive integer and real values only. It controls the writing of the optional plus sign when the edit descriptor `I`, `F`, `E`, `EN`, `ES`, `D`, or `G` is used. The sign edit descriptors are as follows:

Descriptor	Meaning
`SP`	The plus sign is always written
`S`, `SS`	The plus sign is not written

The descriptors remain in effect until another sign edit descriptor is encountered in the format specification. The descriptors have no effect during formatted input data transfers.

For example, assume in the following code that `SPEED(1)` and `SPEED(2)` are `1.46` and `2.3412` respectively:

```
      WRITE(6, 110)  (SPEED(K), K = 1, 2)
  110 FORMAT(SP, 2F10.2)
```

The preceding code produces the following:

bbbbb `+1.46`*bbb* `+234.12`

Scale factors

10.9.5

The *k*`P` edit descriptor indicates scaling, where the scale factor *k* is a signed integer literal constant.

The scale factor is zero at the beginning of a formatted I/O statement. When a *k*`P` descriptor occurs, the scale factor becomes *k*, and all succeeding numeric fields processed with an `F`, `E`, `EN`, `ES`, `D`, or `G` edit descriptor may be affected by this scale factor until another *k*`P` edit descriptor occurs.

On input:

- If the input field has no exponent, the scale factor effect is that the external number equals the internal number multiplied by a scale factor 10^k.
- The scale factor has no effect if the input field has an exponent.

Section 10

Input example:

Assume that the input record contains `10.12`:

```
      READ (5,100) MASS
  100 FORMAT (3PF15.3)
```

The preceding code gives `MASS` the value `10120.0`.

On output:

- For the `F` edit descriptor, the value is multiplied by 10^k.
- For the `E` and `D` edit descriptors, the significand of the value is multiplied by 10^k and the exponent is reduced by k.
- The `G` edit descriptor is not affected by the scale factor if the number will print correctly with the appropriate `F` edit descriptor as described in Table 36, page 486. Otherwise, the scale factor for the `G` edit descriptor has the same effect as for the `E` edit descriptor.
- `EN` and `ES` edit descriptors are not affected by a scale factor.

Assume that `TREE` has the value `12.96`:

```
      WRITE(6,200) TREE
  200 FORMAT(2PG10.1)
```

The preceding code produces the following:

```
b1296.E-02
```

Blanks in numeric fields

10.9.6

Blanks other than leading blanks are ignored or interpreted as zero characters in numeric input fields as determined by the blank edit descriptors:

Descriptor	Meaning
`BN`	Treat nonleading blanks in numeric input fields as nonexistent
`BZ`	Treat nonleading blanks in numeric input fields as zeros

Section 10

The interpretation is for input fields only when the field is processed using an I, B, O, Z, F, E, EN, ES, D, or G edit descriptor; output fields are not affected. The BLANK= specifier in the OPEN statement affects the interpretation of blanks if no BN or BZ descriptor is used.

On input:

- The BLANK= specifier for an internal file is NULL.
- If the BLANK= specifier is NULL, or a BN edit descriptor is in effect, the nonleading blanks are ignored in succeeding numeric fields.
- If the BLANK= specifier is ZERO, or a BZ edit descriptor is in effect, the nonleading blanks are interpreted as zeros in succeeding numeric fields.
- The BN and BZ edit descriptors override the effect of the BLANK= specifier during the execution of a particular input data transfer statement.

Example:

```
      READ (5, 100)  N1, N2
  100 FORMAT (I5, BZ, I5)
```

Considering the preceding code, if the input record is as follows, and unit 5 has been opened with a BLANK= specifier equal to NULL, the values assigned to N1 and N2 are 99 and 90909, respectively:

*b*9*b*9*b*9*b*9*b*9

Character string edit descriptors

10.10

Character string edit descriptors are used to transfer characters to an output record. They must not be used on input. The character string edit descriptors are apostrophe, quote, and Hollerith. They have the following format:

' *characters* '

Section 10

" *characters* *"*

c `H` *characters*

On output:

- The apostrophe and quote edit descriptors have the form of literal character constants with no kind parameter values. They cause those constants to be placed in the output.
- The Hollerith descriptor *c*`H . . .` may be used to print the *c* characters following the `H`. *c* must be a positive integer and must not have a kind value specified for it.
- To write a quote in the output field when a quote is the delimiting character , use two consecutive quotes; to write an apostrophe in the output field when an apostrophe is the delimiting character, use two consecutive apostrophes.
- The field width is the length of the character constant, but doubled apostrophes or quotes count as one character.

For example, assume in the following code that `TEMP` has the value `32.120001`:

```
      WRITE (6, 120)  TEMP
  120 FORMAT (' TEMPERATURE = ', F13.6)
```

The preceding code produces the following record:

b`TEMPERATURE`*b*=*bbbbb* `32.120001`

Section 10

List-directed formatting

10.11

List-directed formatting is one of the implicit formatting methods in Fortran. Conversion from characters in `READ` statements to characters in `PRINT` and `WRITE` statements does not use an explicit format specification. The editing occurs based on the type of the list item. Data is separated by commas or blanks. The I/O statement uses an asterisk (`*`) instead of an explicit format specification.

```
READ (5, *)  HOT, COLD, WARM
```

For both input and output:

- A list-directed record consists of values and value separators.
- The values allowed in a list-directed input record are as follows:

Value	Content
null	A null value. Specified, for example, by two consecutive commas (`,,`).
c	Either 1) A literal constant or 2) a nondelimited character string with no embedded blanks. See subsection 10.11.1.1, page 502, for the requirements for nondelimited character constants.
*r*c*	*r* repetitions of the constant *c*. *r* must be an unsigned, nonzero integer.
*r**	*r* repetitions of the null value, where *r* is nonzero. *r* must be an unsigned, nonzero integer.

Embedded blanks are allowed only within values as specified for each data type. For example, embedded blanks are allowed within a delimited character constant or at certain positions within a complex value.

Section 10

- The value separators allowed in a list-directed input record are as follows:

Separator	Meaning
,	A comma, optionally preceded or followed by contiguous blanks.
/	A slash, optionally preceded or followed by contiguous blanks.
<blank>	One or more contiguous blanks between two nonblank values or following the last nonblank value, where a nonblank value is a constant, an *r*c* form, or an *r** form.

- List-directed formatting must not be specified for direct access or nonadvancing sequential data transfer.

- If there are no list items and there is no current record, an input record is skipped or an output record that is empty is written. If there are no list items and there is a current record, the current record is skipped (the file is positioned at the end of the current record) or the current record is terminated at the current position. Recall that a current record exists only if the previous I/O data transfer to the unit was nonadvancing. An empty record is either a blank-filled record or a record with no characters in it, depending on the file structure used. If you are using a Cray PVP system or a CRAY T3E system, see the *Application Programmer's I/O Guide* , publication SG–2168, for more information on file structures.

- The end of a record has the same effect as a blank, unless it occurs within a delimited character literal constant. Similarly, a sequence of two or more consecutive blanks is treated as a single blank.

10.11.1 List-directed input

Input values are generally accepted as list-directed input if they are the same as those required for explicit formatting with an edit descriptor. The exceptions are as follows:

- When the data list item is of type integer, the constant must be of a form suitable for the `I` edit descriptor. The CF90 compiler permits binary, octal, and hexadecimal based values in a list-directed input record to correspond to `I` edit descriptors.

Section 10

ANSI/ISO: The Fortran 90 standard specifies that binary, octal, and hexadecimal values must not be used in a list-directed input record.

- When the data list item is of type real, the constant must be of a form suitable for the `F` edit descriptor. If no decimal point appears in the constant, the constant has no fractional digits specified for it.
- Blanks are never zeros.
- Embedded blanks are allowed within a delimited character constant. Values of type complex include the parentheses for a complex constant, and blanks may occur before or after the comma, and before or after the parentheses.

```
"NICE DAY"
(1.2, 5.666 )
TODAY
```

- Logical items must not use value separators as the optional characters following the `T` or `F`. `TUESDAY` is allowed; `T,TOO` is not because the value `TOO` will be used as the next input value.
- An end-of-record cannot occur within a constant, except a complex constant or a delimited character constant. For a complex constant, the end of record can occur between the real part and the comma, or between the comma and the imaginary part. For a character constant, the end-of-record can occur anywhere in the constant except between any consecutive (doubled) quotes or apostrophes in the constant. The end-of-record does not cause a blank or any other character to become part of the character value. A complex or character constant can be continued on as many records as needed.
- Value separators may appear in any delimited character constant. They are, however, not interpreted as value separators, but are characters in the delimited character constant.
- Assume that *len* is the length of the corresponding input list item and *w* is the number of effective characters in the character value. If $len \le w$, the leftmost *len* characters of the constant are used. If $len > w$, the *w* characters of the constant are left justified in the input list item and the list item is blank filled on the right.

For example, consider the following code:

```
CHARACTER (2) NAME
   . . .
READ (5,*)  NAME
```

Assume that the input record is as follows:

```
JONES
```

After the `READ` statement, the value in `NAME` is `JO`, because *len* (=2) is less than *w* (=5).

Requirements for nondelimited character strings as values

10.11.1.1

In certain cases, the delimiters are not required for character values on input. However, nondelimited character strings impose the following requirements:

- The corresponding data list item must be of type character.
- The character string must not contain any value separator.
- The character string must not be continued across a record boundary.
- The first nonblank character is neither an apostrophe (`'`) nor a quote (`"`).
- The leading characters are not a string of digits followed immediately by an asterisk.

In any of these cases, the character constant represented by the character string is terminated by the first value separator or end-of-record, and apostrophes (`'`) and quotes (`"`) are not doubled.

Null values
10.11.1.2

Null values are used to specify no change of the items in the input item list. Null values have the following forms:

- No value between separators, such as `,,`
- A nonblank value separator as the first entity in the record; for example, a record beginning with slash as the first nonblank character represents a null value, as in `/4.56`.
- *r** followed by a value separator as in the following:

```
7*,'TODAY'
```

An end-of-record does not signify a null value.

The null value does not affect the definition status or value of the corresponding list item.

For a complex constant, the entire constant can be null, but not one of the parts.

If a slash terminates input, the remaining characters in the record are ignored, and the remaining list items are treated as though null values had been read. This applies to any remaining items in an implied-`DO` or to any remaining elements of an array.

```
REAL AVERAGE(2)
READ (5, *) NUMBER, AVERAGE
```

If the input record is the following, the result is that `NUMBER = 6`, `AVERAGE(1)` is unchanged, and `AVERAGE(2) = 2.418`:

```
b6,,2.418
```

List–directed output
10.11.2

List-directed output uses similar conventions to those used for list-directed input. The following rules and restrictions apply to both input and output:

- Except for nondelimited character values, a comma, optionally preceded or followed by a blank, is used as a separator.

- New records are begun as needed, at any point in the list of output items. A new record does not begin in the middle of a value, except as noted below for character values. Each new record begins with a blank for carriage control, except for delimited character constants.
- Slashes and null values are not written.
- If two or more consecutive values from an array are identical, they can be written using the repeat factor *r*c*.

There are a few exceptions. These are noted as follows for each of the intrinsic types:

- Integer . The effect is as though an `I`*w* edit descriptor were used, using a suitable value for *w*.
- Real. The effect is as though an `0PF`*w*`.`*d* or an `1PE`*w*`.`*d*`E`*e* edit descriptor were used, using suitable values for *w*, *d*, and *e*. The edit descriptor chosen depends on the magnitude of the number written.
- Complex. The real and imaginary parts are enclosed in parentheses and separated by a comma. A complex value must fit within one record.
- Logical. List-directed output prints `T` or `F` depending on the value of the logical data item.
- Character . The form of the output for character values depends on the value of the `DELIM=` specifier in the `OPEN` statement for that unit.

  The following are true if there is no `DELIM=` specifier, if the value of the `DELIM=` specifier is `NONE`, or if the file is an internal file:

  – Character values are not delimited

  – Character values are not surrounded by value separators

  – Only one quote or apostrophe is needed for each quote or apostrophe in the string transferred

  – A blank is inserted at the beginning of new records for a continued character value

The following are true if the DELIM= specifier is QUOTE or APOSTROPHE:

– Character values are delimited with the specified delimiter

– All values are surrounded by value separators

– A character that is the same as the specified delimiter is doubled when written to the output record

– No blank is inserted at the beginning of a continued record for carriage control in the case of a character value continued between records

List-directed output of real values uses either an F or an E format with a number of decimal digits of precision that assures full-precision printing of the real values. This allows later formatted or list-directed input of real values to result in the generation of a bit-identical binary floating-point representation, which allows a value to be written and reread without changing the stored value.

You can set the LISTIO_PRECISION environment variable to control the number of digits of precision printed by list-directed output. The following values can be assigned to LISTIO_PRECISION:

Value	Result
FULL	Prints full precision (default).
PRECISION	Prints *x* or *x*+1 decimal digits, where *x* is value of the PRECISION intrinsic function for a given real value. This is a smaller number of digits, which usually ensures that the last decimal digit is accurate to within 1 unit. This number of digits is usually insufficient to assure that subsequent input will restore a bit-identical floating-point value.
YMP80	Causes list-directed and namelist output of real values to be of the format used in UNICOS 8.0 and previous library versions on CRAY Y-MP systems. The YMP80 specification is available in UNICOS 8.0 libraries on Cray PVP systems, but this functionality will be removed in a later UNICOS release.

ANSI/ISO: The Fortran 90 standard does not specify full-precision printing or environment variables.

Example:

```
REAL                       :: TEMPERATURE = -7.6
INTEGER                    :: COUNT = 3
CHARACTER(*), PARAMETER :: PHRASE = "This isn't so"

OPEN(10, DELIM = 'NONE')
WRITE(10, *)  TEMPERATURE, COUNT, PHRASE
```

The output record on unit 10 would be:

```
-7.6    3     This isn't so
```

Namelist formatting

10.12

In some programs, it is convenient to create a list of variables that can be read or written by referencing the name of the list. The term *namelist* denotes this kind of data transfer. Before input or output can begin using this facility, the `NAMELIST` specification statement is used to define the list and give it a *group name*. In the following example, the `NAMELIST` statement defines the group name `MEETING` made up of the data objects `JAKE`, `JOE`, and `JANE`.

```
NAMELIST /MEETING/ JOE, JAKE, JANE
```

The namelist input and output records consist of an ampersand (`&`) followed by a namelist group name followed by a sequence of name-value pairs followed by a slash (`/`). A *name-value pair* is a name or a subobject designator , an equal sign, and one or more values separated by value separators; that is, *name*=*value* or *name*=*value,value,* ... . The *name* in a name-value pair must appear in the `NAMELIST` statement specifying the namelist group name. The name-value pairs provide a convenient form of documentation.

For example, consider the following input data transfer statement:

```
READ (*, NML = MEETING)
```

The preceding statement sets the variables `JAKE` and `JOE` when the input record is as follows:

```
&MEETING  JAKE = 3505,  JOE = 1 /
```

It does not change the value of the variable JANE.

Namelist output is convenient for debugging or writing values in a form that can later be read by a READ statement referencing the same namelist group name. All variables in the namelist group name appear in the output record and they must all be defined when the output statement is executed. For example, consider the following WRITE statement:

```
WRITE (*, NML = MEETING)
```

The preceding statement creates the following output record (assuming JANE is defined with the value 0):

```
&MEETING  JAKE = 3505, JOE = 1, JANE = 0 /
```

In namelist I/O records, blanks can appear before the ampersand, after the namelist group name, before or after the equal sign in the name-value pairs, or before the slash terminating the namelist records.

The rules and restrictions for a blank and an end-of-record in a character constant are the same as for delimited character constants in list-directed formatting. Nondelimited character strings are not permitted in namelist input records.

Namelist input

10.12.1

Namelist input consists of the following items, in order:

- An ampersand (&).
- The group name.
- One or more blanks.
- A sequence of zero or more name-value pairs whose names are in the namelist group and whose values are described in the following subsections. These must be separated by value separators and followed by a slash, which terminates the namelist input.

The name-value pairs are separated by value separators that are composed of the following characters:

- A comma, optionally preceded or followed by contiguous blanks

- A slash, optionally preceded or followed by contiguous blanks
- One or more contiguous blanks between two name-value pairs

Blanks can precede the ampersand or the slash.

When the name in the name-value pair is a subobject designator, it must not be a zero-sized array, zero-sized array section, or a zero-length character string.

A lowercase letter is the same as an uppercase letter and vice versa when used in the group name.

An example of namelist input is as follows:

```
READ (*, NML = MEETING)
```

The input record for the preceding statement might be as follows:

```
&MEETING JAKE = 3500, JOE = 100, JANE = 0/
```

Names in name-value pairs
10.12.1.1

This subsection describes the rules and restrictions for the names used in the name-value pairs in the namelist records.

- The name-value pairs can appear in any order in the input records.
- The name-value pairs are evaluated serially, in left-to-right order.
- A name in the namelist group can be omitted.
- Each name must correspond with a name in the designated namelist group; a component name, if any, must also be the name of a component of a structure named in the namelist group.
- Optionally-signed integer literal constants with no kind parameter values must be used in all expressions that appear in subscripts, section designators, or substring designators.
- The name of a structure or a subobject designator can be the name in a name-value pair.

- A name in an input record must not contain embedded blanks. A name in the name-value pair can be preceded or followed by one or more blanks.
- A lowercase letter is the same as an uppercase letter and vice versa when used in the name of a name-value pair.
- A namelist group object name or subobject designator can appear in more than one name-value pair in a sequence.

Recall that each name must not be the name of an array dummy argument with nonconstant bounds, an automatic object, a character variable with nonconstant length, a pointer, or a structure variable of a type with a component that is a pointer, or an allocatable array.

Values in name-value pairs 10.12.1.2

The value in a name-value pair must be in a form acceptable for a format specification for the type of the name, except for restrictions noted below.

Null values can have the following forms:

- No value between value separators
- No value between the equal sign and the first value separator
- The *r** form, followed by one or more blanks

Null values do not change the value of the named item or its definition status. An entire complex constant can be null; neither of the parts can be. The end of a record following a value separator does not specify a null value.

Each value is a null value or one of the following forms; in these forms, *r* is assumed to be a repeat factor and is nonzero:

Value	Representation
c	Indicates a literal constant, which must be unsigned
*r*c*	Indicates *r* successive literal constants *c*
*r**	Indicates *r* successive null values

Section 10

The form of a value must be acceptable to a format specification for an entity of the type of the corresponding list item, except as noted below; for example, the value *c* corresponding to a list item of type real can be of the following forms:

```
1
1.0E0
```

The value *c* corresponding to a list item of type real cannot be of the following forms:

```
(1.0,0.0)
A0
1.0EN-3
2.2_QUAD
```

Blanks are never treated as zeros, and embedded blanks must not appear in numeric or logical constants. The exception is that a blank can appear as a character in a character constant, or preceding or following the real or imaginary parts of a complex constant.

The number of values following the equal sign must not be larger than the number of elements of the array when the name in the name-value pair is an array, or must not be larger than the ultimate number of components when the name in the name-value pair is that of a structure. Any array or component that is an array is filled in array element order.

Consider the following example:

```
TYPE PERSON
     INTEGER LEN
     CHARACTER(10) :: NAME
END TYPE PERSON

TYPE(PERSON) PRESIDENT, VICE_PRES

NAMELIST /PERSON_LIST/ PRESIDENT, VICE_PRES

READ (5, NML = PERSON_LIST)
```

The input record for the preceding code might be as follows:

```
&PERSON_LIST PRESIDENT%LEN=10,
PRESIDENT%NAME="WASHINGTON",
                VICE_PRES%LEN=5, VICE_PRES%NAME="ADAMS"/
```

If there are fewer values in the expanded sequence than array elements or structure components, null values are supplied for the missing values.

If a slash occurs in the input, it is as if null values were supplied for the remaining list items, and the namelist input data transfer is terminated. The remaining values after the slash within the current record are ignored. The namelist input record can be terminated by the characters `&END`.

An integer value is interpreted as if the data edit descriptor were `I`*w* for a suitable value of *w*.

A complex value consists of a pair of parentheses surrounding the real and imaginary parts, separated by a comma. Blanks can appear before and after these values.

A logical value must not contain slashes, commas, or blanks as part of the optional characters permitted for logical values. The required characters for logical values include the first few characters, such as the initial:

- `.T`
- `.F`
- `T`
- `F`

The optional characters on logical values, which include `ALSE.`, `RUE.`, and so on, cannot include slashes, commas, or blanks.

A character literal constant may contain slashes, commas, or blanks as part of the constant. On namelist input, the `DELIM=` specifier is ignored.

Blanks
10.12.1.3

Blanks are part of the value separator except when used in the following situations:

- In a character constant
- Before or after the parts of a complex constant
- Before or after an equal sign, unless after the equal sign the blanks are followed immediately by a comma or slash
- Before the ampersand indicating the namelist group name and after the namelist group name

Use of namelist input
10.12.1.4

Namelist input requires the namelist group name, preceded by an ampersand, to be on the first nonblank record read by the namelist READ statement.

The following is an example of namelist input:

```
REAL A(3), B(3)
CHARACTER(LEN = 3) CHAR
COMPLEX X
LOGICAL LL
NAMELIST /TOKEN/ I, A, CHAR, X, LL, B
READ (*, NML = TOKEN)
```

Assume that the input record for the preceding is as follows:

```
&TOKEN A(1:2) = 2*1.0  CHAR = "NOP" B = ,3.13,,
       X = (2.4,0.0)  LL = T /
```

The results of the READ statement are as follows:

Name	Value
I	Unchanged
A(1)	1.0
A(2)	1.0

Section 10

Name	Value
`A(3)`	Unchanged
`B(1)`	Unchanged
`B(2)`	3.13
`B(3)`	Unchanged
`CHAR`	`"NOP"`
`X`	(2.4, 0.0)
`LL`	True

Namelist output
10.12.2

Value separators can be blanks, commas, or a combination of blanks and commas. A new record can begin anywhere, except within a name or value, unless the value is a character constant or a complex constant; a record can begin anywhere within a character constant, or can begin before or after the comma, or left or right parenthesis of a complex constant. A blank can occur anywhere, except in a name or a noncharacter value.

Use of namelist output
10.12.2.1

A number of rules, similar to those for list-directed formatting, apply for namelist output. They are as follows:

- Namelist output consists of a series of records. The first nonblank record begins with an ampersand, followed by the namelist group name, followed by a sequence of name-value pairs, one pair for each variable name in the namelist group object list of the `NAMELIST` statement, and ends with a slash.
- A logical value is either `T` or `F`.
- An integer value is one that would be produced by an `I`*w* edit descriptor using a suitable value of *w*.
- For real output, the rules for list-directed output are followed using reasonable values for the *w*, *e*, and *d* that appear in real data edit descriptors and are appropriate for the output value.

Section 10

- Parentheses enclose the value of a complex constant, and the parts are separated by a comma. A complex value must fit within one record. Blanks can be embedded after the comma and before the end of the record.
- Character constants follow the rules for list-directed output.
- Repeat factors of the form *r*c* are allowed on output for successive identical values.
- An output record does not contain null values.
- Each record begins with a blank for carriage control.
- No values are written for zero-sized arrays.
- For zero-length character strings, the name is written, followed by an equal sign, followed by a zero-length character string, and followed by a value separator or slash.

```
          NAMELIST /CALC/ DEPTH, PRESSURE
          DIMENSION DEPTH(3), PRESSURE(3)
          FLOAT1 = 0.2
          FLOAT2 = 3.0
          DO 10 I=1,3
            DEPTH(I) = FLOAT1 + FLOAT(I)
            PRESSURE(I) = FLOAT2 + (FLOAT(I-1) * 0.1)
   10     CONTINUE
          WRITE(6,NML=CALC)
          END
```

This program produces the following output:

```
&CALC  DEPTH = 1.2, 2.2, 3.2,  PRESSURE = 3., 3.1, 3.2 /
```

10.12.2.2 DELIM= specifier for character constants

The form of the output for character values depends on the value of the DELIM= specifier in the OPEN statement for that unit.

Section 10

If there is no `DELIM=` specifier or if the value of the `DELIM=` specifier is `NONE`, the following actions occur:

- Character values are not delimited.
- Character values are not surrounded by value separators.
- Only one quote or apostrophe is needed for each quote or apostrophe in the string transferred.
- A blank is inserted in new records for a continued character value to allow for carriage control.

If the `DELIM=` specifier is `QUOTE` or `APOSTROPHE`, the following actions occur:

- Character values are delimited with the specified delimiter.
- A character that is the same as the specified delimiter is doubled when written to the output record.

For example, in the following code, assume `LEFT` and `RIGHT` with values `"SOUTH"` and `"NORTH"`. The program has a `DELIM=` specifier of `QUOTE` for unit 10:

```
CHARACTER(5) LEFT, RIGHT
NAMELIST /TURN/ LEFT,RIGHT
LEFT = 'SOUTH'
RIGHT = 'NORTH'
WRITE(10,NML=TURN)
END
```

The preceding code produces the following output record:

```
&TURN  LEFT = SOUTH, RIGHT = NORTH /
```

Note that if the `DELIM=` specifier is `NONE` in an `OPEN` statement, namelist output might not be usable as namelist input when character values are transferred, because namelist input of character values requires delimited character values.

Namelist distinctions
10.12.3

The CF90 compiler has extended the namelist feature. The following additional rules govern CF90 namelist processing:

- An ampersand (`&`) or dollar sign (`$`) can precede the namelist group name or terminate namelist group input. If an ampersand precedes the namelist group name, either the slash (`/`) or the ampersand must terminate the namelist group input. If the dollar sign precedes the namelist group name, either the slash or the dollar sign must terminate the namelist group input.

- Octal and hexadecimal constants are allowed as input to integer and single-precision real namelist group items. An error is generated if octal and hexadecimal constants are specified as input to character, complex, or double-precision real namelist group items.

  Octal constants must be of the following form:

  – `O"123"`

  – `O'123'`

  – `o"123"`

  – `o'123'`

  Hexadecimal constants must be of the following form:

  – `Z"1a3"`

  – `Z'1a3'`

  – `z"1a3"`

  – `z'1a3'`

  ANSI/ISO: The Fortran 90 standard does not specify the namelist distinctions described in this subsection.

Glossary

argument keyword

The name of a dummy (or formal) argument. This name is used in the subprogram definition; it also may be used when the subprogram is invoked to associate an actual argument with a dummy argument. Using argument keywords allows the actual arguments to appear in any order. The Fortran 90 standard specifies argument keywords for all intrinsic procedures. Argument keywords for user-supplied external procedures may be specified in a procedure interface block.

array

(1) A data structure that contains a series of related data items arranged in rows and columns for convenient access. The C shell and the `awk`(1) command can store and process arrays. (2) In Fortran 90, an object with the `DIMENSION` attribute. It is a set of scalar data, all of the same type and type parameters. The rank of an array is at least 1, and at most 7. Arrays may be used as expression operands, procedure arguments, and function results, and they may appear in input/output (I/O) lists.

association

An association permits an entity to be referenced by different names in a scoping unit or by the same or different names in different scoping units. Several kinds of association exist. The principal kinds of association are pointer association, argument association, host association, use association, and storage association.

Autotasking

A trademarked process of Cray Research that automatically divides a program into individual tasks and organizes them to make the most efficient use of the computer hardware.

CIV

A constant increment variable is a variable that is incremented only by a loop invariant value (for example, in a loop with index J, the statement J = J + K, in which K can be equal to 0, J is a CIV).

constant

A data object whose value cannot be changed. A named entity with the `PARAMETER` attribute is called a named constant. A constant without a name is called a literal constant.

construct

A sequence of statements that starts with a `SELECT CASE`, `DO`, `IF`, or `WHERE` statement and ends with the corresponding terminal statement.

control construct

An action statement that can change the normal execution sequence (such as a `GO TO`, `STOP`, or `RETURN` statement) or a `CASE`, `DO`, or `IF` construct.

data entity

A data object, the result of the evaluation of an expression, or the result of the execution of a function reference (also called the function result). A data entity always has a type.

data object

A constant, a variable, or a part of a constant or variable.

declaration

A nonexecutable statement that specifies the attributes of a data object (for example, it may be used to specify the type of a variable or function result or the shape of an array).

definition

This term is used in two ways. (1) A data object is said to be defined when it has a valid or predictable value; otherwise, it is undefined. It may be given a valid value by execution of statements such as assignment or input. Under certain circumstances, it may subsequently become undefined. (2) Procedures and derived types are said to be defined when their descriptions have been supplied by the programmer and are available in a program unit.

derived type

A type that is not intrinsic (a user-defined type); it requires a type definition to name the type and specify its components. The components may be of intrinsic or user-defined types. An object of derived type is called a structure. For each derived type, a structure constructor is available to specify values. Operations on objects of derived type must be defined by a function with an interface and the generic specifier `OPERATOR`. Assignment for derived type objects is defined intrinsically, but it may be redefined by a subroutine with the `ASSIGNMENT` generic specifier. Data objects of derived type may be used as procedure arguments and function results, and they may appear in input/output (I/O) lists.

designator

Sometimes it is convenient to reference only part of an object, such as an element or section of an array, a substring of a character string, or a component of a structure. This requires the use of the name of the object followed by a selector that selects a part of the object. A name followed by a selector is called a *designator* .

entity

(1) In Open Systems Interconnection (OSI) terminology, a layered protocol machine. An entity in a layer performs the functions of the layer in one computer system, accessing the layer entity below and providing services to the layer entity above at local service access points. (2) In Fortran 90, a general term used to refer to any Fortran 90 concept (for example, a program unit, a common block, a variable, an expression value, a constant, a statement label, a construct, an operator, an interface block, a derived type, an input/output (I/O) unit, a name list group, and so on).

executable construct

A statement (such as a `GO TO` statement) or a construct (such as a `DO` or `CASE` construct).

expression

A set of operands, which may be function invocations, and operators that produce a value. See also *regular expression*.

generic specifier

An optional component of the `INTERFACE` statement. It can take the form of an identifier, an `OPERATOR (`*defined_operator*`)` clause, or an `ASSIGNMENT (=)` clause.

intrinsic

Anything that the language defines is intrinsic. There are intrinsic data types, procedures, and operators. You may use these freely in any scoping unit. Fortran programmers may define types, procedures, and operators; these entities are not intrinsic.

multitasking

(1) The parallel execution of two or more parts of a program on different CPUs; these parts share an area of memory. (2) A method in multiuser systems that incorporates multiple interconnected CPUs; these CPUs run their programs simultaneously (in parallel) and share resources such as memory, storage devices, and printers. This term can often be used interchangeably with *parallel processing*.

name

A term that identifies many different entities of a program such as a program unit, a variable, a common block, a construct, a formal argument of a subprogram (dummy argument), or a user-defined type (derived type). A name may be associated with a specific constant (named constant).

operator

(1) A symbolic expression that indicates the action to be performed in an expression; operator types include arithmetic, relational, and logical. (2) In Fortran 90, an operator indicates a computation that involves one or two operands. Fortran 90 defines several intrinsic operators (for example, `+`, `-`, `*`, `/`, `**` are numeric operators, and `.NOT.`, `.AND.`, and `.OR.` are logical operators). Users also may define operators for use with operands of intrinsic or derived types.

overindexing

The nonstandard practice of referencing an array with a subscript not contained between the declared lower and upper bounds of the corresponding dimension for that array. This practice sometimes, but not necessarily, leads to referencing a storage location outside of the entire array.

parallel processing

Processing in which multiple processors work on a single application simultaneously.

procedure

(1) A named sequence of control statements and/or data that is saved in a library for processing at a later time, when a calling statement activates it; it provides the capability to replace values within the procedure. (2) In Fortran 90, procedure is defined by a sequence of statements that expresses a computation that may be invoked as a subroutine or function during program execution. It may be an intrinsic procedure, an external procedure, an internal procedure, a module procedure, a dummy procedure, or a statement function. If a subprogram contains an `ENTRY` statement, it defines more than one procedure.

procedure interface

In Fortran 90, a sequence of statements that specifies the name and characteristics of one or more procedures, the name and attributes of each dummy argument, and the generic specifier by which it may be referenced if any. See *generic specifier.*

In FORTRAN 77 and Fortran 90, a *generic function* is one whose output value data type is determined by the data type of its input arguments. In FORTRAN 77, the only generic functions allowed are those that the standard defines. In Fortran 90, programmers may construct their own generic function by creating a generic interface, which is like a regular procedure interface, except that it has a generic specifier (the name of the generic function) after the keyword `INTERFACE`.

reference

A data object reference is the appearance of a name, designator, or associated pointer in an executable statement that requires the value of the object. A procedure reference is the appearance of the procedure name, operator symbol, or assignment symbol in an executable program that requires execution of the procedure. A module reference is the appearance of the module name in a `USE` statement.

regular expression

(1) In POSIX, a pattern (sequence of characters or symbols) constructed according to the defined rules or defined RE rules (see *regular expression rules* in the online glossary). (2) A character string that contains operating system wildcards (metacharacters) that can match multiple substitution characters. Such a string will match more than one literal string; it is used to filter a specified string or type of string from an input of multiple strings. The main editors in UNIX, `ed` and `vi`, use regular expressions.

scalar

(1) In Fortran 90, a single object of any intrinsic or derived type. A structure is scalar even if it has a component that is an array. The rank of a scalar is 0. (2) A nonvectorized, single numerical value that represents one aspect of a physical quantity and may be represented on a scale as a point. This term often refers to a floating-point or integer computation that is not vectorized; more generally, it also refers to logical and conditional (jump) computation.

scoping unit

Part of a program in which a name has a fixed meaning. A program unit or subprogram generally defines a scoping unit. Type definitions and procedure interface bodies also constitute scoping units. Scoping units do not overlap, although one scoping unit may contain another in the sense that it surrounds it. If a scoping unit contains another scoping unit, the outer scoping unit is referred to as the host scoping unit of the inner scoping unit.

sequence

A set ordered by a one-to-one correspondence with the numbers 1, 2, through n. The number of elements in the sequence is n. A sequence may be empty, in which case, it contains no elements.

shell variable

A name representing a string value. Variables that are usually set only on a command line are called *parameters* (positional parameters and keyword parameters). Other variables are simply names to which a user (user -defined variables) or the shell itself may assign string values. The shell has predefined shell variables (for example, `HOME`). Variables are referenced by prefixing the variable name by a `$` (for example, `$HOME`).

statement keyword

A keyword that is part of the syntax of a statement. Each statement, other than an assignment statement and a statement function definition, begins with a statement keyword. Examples of these keywords are `IF`, `READ`, and `INTEGER`. Statement keywords are not reserved words; you may use them as names to identify program elements.

stripmining

An optimization technique used on vectorizing machines in which a vector or array is broken down into chunks the size of the machine's vector register set for simultaneous processing.

subobject

Parts of a data object may be referenced and defined separately from other parts of the object. Portions of arrays are array elements and array sections. Portions of character strings are substrings. Portions of structures are structure components. Subobjects are referenced by designators and are considered to be data objects themselves.

TKR

An acronym that represents attributes for argument association. It represents the data type, kind type parameter, and rank of the argument.

type parameter

Two type parameters exist for intrinsic types: kind and length. The kind type parameter `KIND` indicates the decimal range for the integer type, the decimal precision and exponent range for the real and complex types, and the machine representation method for the character and logical types. The length type parameter `LEN` indicates the length of a character string.

variable

(1) A name that represents a string value. Variables that usually are set only on a command line are called parameters. Other variables are simply names to which the user or the shell may assign string values. (2) In Fortran 90, a data object whose value can be defined and redefined. A variable may be a scalar or an array. (3) In the shell command language, a named parameter. See also *shell variable*.

Index

A

Section 2

D

Section 2

E

Section 2

F

G

H

I

Section 2

S

T

Section 2

Reader's Comment Form

CF90 Fortran Language Reference Manual, Volume 1 SR–3902 2.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to complete the following items, and use the blank space for additional comments.

List the operating systems and programming languages you have used and the years of experience with each.

Your experience with computer systems: _____0-1 year _____1-5 year _____5+years

How did you use this manual: _____in a class _____as a tutorial or introduction _____as a procedural guide _____as a reference _____for troubleshooting _____other

Please rate this manual on the following criteria:

	Excellent			Poor
Accuracy	4	3	2	1
Appropriateness (correct technical level)	4	3	2	1
Accessibility (ease of finding information)	4	3	2	1
Physical qualities (binding, printing, illustrations)	4	3	2	1
Terminology (correct, consistent, and clear)	4	3	2	1
Number of examples	4	3	2	1
Quality of examples	4	3	2	1
Index	4	3	2	1

Please use the space below for your comments about this manual. Please include general comments about the usefulness of this manual. If you have discovered inaccuracies or omissions, please specify the number of the page on which the problem occurred.

Name ______________________	Address ______________________
Title ______________________	City ______________________
Company ______________________	State/Country ______________________
Telephone ______________________	Zip code ______________________
Today's date ______________________	Electronic mail address ______________________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

ATTN: Software Publications Group
655 LONE OAK DR BLDG F
EAGAN MN 55121-9957

Fold

Cut along this line

Reader's Comment Form

CF90 Fortran Language Reference Manual, Volume 1 SR–3902 2.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to complete the following items, and use the blank space for additional comments.

List the operating systems and programming languages you have used and the years of experience with each.

Your experience with computer systems: _____0-1 year _____1-5 year _____5+years

How did you use this manual: _____in a class _____as a tutorial or introduction _____as a procedural guide _____as a reference _____for troubleshooting _____other

Please rate this manual on the following criteria:

	Excellent			Poor
Accuracy	4	3	2	1
Appropriateness (correct technical level)	4	3	2	1
Accessibility (ease of finding information)	4	3	2	1
Physical qualities (binding, printing, illustrations)	4	3	2	1
Terminology (correct, consistent, and clear)	4	3	2	1
Number of examples	4	3	2	1
Quality of examples	4	3	2	1
Index	4	3	2	1

Please use the space below for your comments about this manual. Please include general comments about the usefulness of this manual. If you have discovered inaccuracies or omissions, please specify the number of the page on which the problem occurred.

Name ________________________________
Title ________________________________
Company ________________________________
Telephone ________________________________
Today's date ________________________________

Address ________________________________
City ________________________________
State/Country ________________________________
Zip code ________________________________
Electronic mail address ________________________________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

ATTN: Software Publications Group
655 LONE OAK DR BLDG F
EAGAN MN 55121-9957

Fold

Cut along this line