PGI® Fortran Reference

Release 2013

The Portland Group®

PGI® Fortran Reference

Printed in the United States of America

First Printing: Release 6.0 March, 2005
Second Printing: Release 6.1 December, 2005
Third Printing: Release 6.1-3 February, 2006
Fourth Printing: Release 7.0-1 December, 2006
Fifth Printing: Release 7.0-2 February, 2007
Sixth Printing: Release 7.2-1 May, 2008
Seventh Printing: Release 7.2-4 August, 2008
Eighth Printing: Release 8.0-1 November, 2008
Ninth Printing: Release 8.1 November, 2008
Tenth Printing: Release 9.0 June, 2009
Eleventh Printing: Release 2010 November, 2009
Twelfth Printing: Release 2010, 10.3 March, 2010
Thirteenth Printing: Release 2010, 10.3 June, 2010
Fourteenth Printing: Release 2011, 11.0 December, 2010
Fifteenth Printing: Release 2011, 11.2 February, 2011
Sixteenth Printing: Release 2012, 12.1 January, 2012
Seventeenth Printing: Release 2012, 12.2 February, 2012
Eighteenth Printing: Release 2012, 12.6 June, 2012
Nineteenth Printing: Release 2012, 12.10 October, 2012
Twentieth Printing: Release 2013, 13.3 March, 2013
Twenty-first Printing: Release 2013, 13.8 August, 2013
Twenty-second Printing: Release 2013, 13.9 September, 2013

www.pgroup.com

Technical support: trs@pgroup.com
Sales: sales@pgroup.com
Web: www.pgroup.com

ID: 13254149

Contents

Figures

Tables

Examples

Preface

This manual describes the Portland Group's implementation of the FORTRAN 77, Fortran 90/95, and Fortran 2003 languages. Collectively, The Portland Group compilers that implement these languages are referred to as the PGI Fortran compilers. This manual is part of a set of documents describing the Fortran language and the compilation tools available from The Portland Group. It presents the Fortran language statements, intrinsics, and extension directives.

The Portland Group's Fortran compilation system includes a compilation driver, multiple Fortran compilers, associated runtime support and mathematical libraries, and associated software development tools for debugging and profiling the performance of Fortran programs. Depending on the target system, The Portland Group's Fortran software development tools may also include an assembler or a linker. You can use these tools to create, debug, optimize and profile your Fortran programs. "Related Publications," on page xix lists other manuals in the PGI documentation set.

Audience Description

This manual is intended for people who are porting or writing Fortran programs using the PGI Fortran compilers. To use Fortran you should be aware of the role of Fortran and of source-level programs in the software development process and you should have some knowledge of a particular system or workstation cluster. To use the PGI Fortran compilers, you need to be familiar with the Fortran language FORTRAN77, Fortran 90/95, or F2003 as well as the basic commands available on your host system.

Compatibility and Conformance to Standards

The PGI Fortran compilers, PGF77, PGF95 and PGFORTAN, run on a variety of 32-bit and 64-bit x86 processor-based host systems. The PGF77 compiler accepts an enhanced version of FORTRAN 77 that conforms to the ANSI standard for FORTRAN 77 and includes various extensions from VAX/VMS Fortran, IBM/VS Fortran, and MIL-STD-1753. The PGF95 compiler accepts a similarly enhanced version of the ANSI standard for Fortran 90/95.

For further information on the Fortran language, you can also refer to the following:

- American National Standard Programming Language FORTRAN, ANSI X3. -1978 (1978).
- ISO/IEC 1539 : 1991, Information technology – Programming Languages – Fortran, Geneva, 1991 (Fortran 90).

- ISO/IEC 1539 : 1997, Information technology – Programming Languages – Fortran, Geneva, 1997 (Fortran 95).
- Fortran 95 Handbook Complete ISO/ANSI Reference, Adams et al, The MIT Press, Cambridge, Mass, 1997.
- Fortran 2003 Handbook, The Complete Syntax, Features and Procedures, Adams et al, Springer; 1st Edition. 2008.
- OpenMP Fortran Application Program Interface, Version 2.0, November 1999, http://www.openmp.org.
- Programming in VAX Fortran, Version 4.0, Digital Equipment Corporation (September, 1984).
- IBM VS Fortran, IBM Corporation, Rev. GC26-4119.
- Military Standard, Fortran, DOD Supplement to American National Standard Programming Language Fortran, ANSI x.3-1978, MIL-STD-1753 (November 9, 1978).

Organization

This manual is divided into the following chapters and appendices:

Chapter 1, "*Language Overview*", provides an introduction to the Fortran language.

Chapter 2, "*Fortran Data Types*", describes the data types supported by PGI Fortran compilers and provides examples using various data types. It also contains information on memory allocation and alignment issue.

Chapter 3, "*Fortran Statements*", briefly describes each Fortran statement that the PGI Fortran compilers accept. Longer descriptions are available for PGI extensions.

Chapter 4, "*Fortran Arrays*", describes special characteristics of arrays in Fortran 90/95.

Chapter 5, "*Input and Output*", describes the input, output, and format statements that allow programs to transfer data to or from files.

Chapter 6, "*Fortran Intrinsics*", lists the Fortran intrinsics and subroutines supported by the PGI Fortran compilers.

Chapter 7, "*Object Oriented Programming*", provides a high-level overview of procedures, functions, and attributes from Fortran 2003 that facilitate an object-oriented approach to programming.

Chapter 8, "*OpenMP Directives for Fortran*", lists the language extensions that the PGI Fortran compilers support.

Chapter 9, "*3F Functions and VAX Subroutines*", describes the functions and subroutines in the Fortran runtime library and discusses the VAX/VMS system subroutines and the built-in functions supported by the PGI Fortran compilers.

Chapter 10, "*Interoperability with C*", describes the pointer types and enumerators available for Fortran interoperability with C.

Hardware and Software Constraints

The PGI compilers operate on a variety of host systems and produce object code for a variety of target systems. Details concerning environment-specific values and defaults and host-specific features or limitations are

presented in the PGI User's Guide, the man pages for each compiler in a given installation, and in the release notes and installation instructions included with all PGI compilers and tools software products.

Conventions

This PGI Fortran Reference manual uses the following conventions:

italic
: is used for commands, filenames, directories, arguments, options and for emphasis.

Constant Width
: is used in examples and for language statements in the text.

[item]
: square brackets indicate optional items. In this case item is optional.

{ item2 | item3}
: braces indicate that a selection is required. In this case, you must select either item2 or item3.

filename ...
: ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.

FORTRAN
: Fortran language statements are shown using upper-case characters and a reduced point size.

<TAB>
: non-printing characters, such as TAB, are shown enclosed in greater than and less than characters and a reduced point size.

§
: this symbol indicates an area in the text that describes a Fortran 90/95 Language enhancement. Enhancements are features that are not described in the ANSI Fortran 90/95 standards.

@
: This symbol indicates an area in the text that describes a FORTRAN 77 enhancement. Enhancements may be VAX/VMS, IBM/VM, or military standard MIL-STD-1753 enhancements.

Related Publications

The following documents contain additional information related to compilers and tools available from The Portland Group, Inc.

- The PGI Compiler User's Guide and the PGI Visual Fortran User's Guide describe the general features and usage guidelines for all PGI compilers, and describes in detail various available compiler options in a user's guide format.
- Fortran 95 Handbook, from McGraw-Hill, describes the Fortran 95 language and the statements, data types, input/output format specifiers, and additional reference material that defines ANSI/ISO Fortran 95.
- Fortran 2003 Handbook, from Springer, provides the complete syntax, features and procedures for Fortran 2003.

- System V Application Binary Interface Processor Supplement by AT&T UNIX System Laboratories, Inc, (available from Prentice Hall, Inc.)
- American National Standard Programming Language Fortran, ANSI x.3-1978 (1978).
- Programming in VAX FORTRAN, Version 4.0, Digital Equipment Corporation (September, 1984).
- IBM VS FORTRAN, IBM Corporation, Rev. GC26-4119.
- Military Standard, FORTRAN, DOD Supplement to American National Standard Programming Language FORTRAN, ANSI X3.-1978, MIL-STD-1753 (November 9, 1978).

Chapter 1. Language Overview

This chapter describes the basic elements of the Fortran language, the format of Fortran statements, and the types of expressions and assignments accepted by the PGI Fortran compilers.

The PGF77 compiler accepts as input FORTRAN 77 and produces as output assembly language code, binary object code or binary executables in conjunction with the assembler, linker and libraries on the target system. The input language must be extended FORTRAN 77 as specified in this reference manual. The PGF95 compiler functions similarly for Fortran 90/95.

This chapter is not an introduction to the overall capabilities of Fortran. Rather, it is an overview of the syntax requirements of programs used with the PGI Fortran compilers. The Fortran 95 Handbook provides details on the capabilities of Fortran 90/95 language. The Fortran 2003 Handbook, provides the complete syntax, features and procedures for Fortran 2003.

Elements of a Fortran Program Unit

A Fortran program is composed of SUBROUTINE, FUNCTION, MODULE, BLOCK DATA, or PROGRAM program units.

Fortran source code consists of a sequence of program units which are to be compiled. Every program unit consists of statements and optionally comments beginning with a program unit statement, either a SUBROUTINE, FUNCTION, or PROGRAM statement, and finishing with an END statement (BLOCK DATA and MODULE program units are also allowed).

In the absence of one of these statements, the PGI Fortran compilers insert a PROGRAM statement.

Statements

Statements are either executable statements or nonexecutable specification statements. Each statement consists of a single line or source record, possibly followed by one or more continuation lines. Multiple statements may appear on a single line if they are separated by a semicolon (;). Comments may appear on any line following a comment character (!).

Free and Fixed Source

Fortran permits two types of source formatting, fixed source form and free source form.

- **Fixed source form** uses the traditional Fortran approach where specific column positions are reserved for labels, continuation characters, and statements and blank characters are ignored. The PGF77 compiler supports only fixed source form. The PGF77 compiler also supports a less restrictive variety of fixed source form called tab source form.
- **Free source form** introduced with Fortran 90 places few restrictions on source formatting; the context of an element, as well as the position of blanks, or tabs, separate logical tokens. You can select free source form as an option to PGF95 or PGFORTRAN in one of these ways:
  - Use the compiler option `-Mfreeform`.
  - Use either the suffix .f90, the suffix .f95, or the suffix .f03.

Statement Ordering

Fortran statements and constructs must conform to ordering requirements imposed by the language definition. Figure 1.1, "Order of Statements" illustrates these requirements. Vertical lines separate statements and constructs that can be interspersed. Horizontal lines separate statements that must not be interspersed.

These rules are less strict than those in the ANSI standard. The differences are as follows:

- DATA statements can be freely interspersed with PARAMETER statements and other specification statements.
- NAMELIST statements are supported and have the same order requirements as FORMAT and ENTRY statements.
- The IMPLICIT NONE statement can precede other IMPLICIT statements.

Figure 1.1. Order of Statements

<table>
<tr><td rowspan="8">Comments
and
INCLUDE
Statements</td><td colspan="4">OPTIONS Statement</td></tr>
<tr><td colspan="4">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statements</td></tr>
<tr><td colspan="4">USE Statements</td></tr>
<tr><td colspan="4">IMPORT Statements</td></tr>
<tr><td rowspan="4">NAMELIST,
FORMAT,
and ENTRY
Statements</td><td colspan="3">IMPLICIT NONE Statements</td></tr>
<tr><td colspan="2">IMPLICIT Statements</td><td rowspan="2">PARAMETER</td></tr>
<tr><td rowspan="2">Data Statements</td><td>Other Specifications</td></tr>
<tr><td colspan="2">Statement Function Definition</td></tr>
<tr><td colspan="4">EXECUTABLE Statements</td></tr>
<tr><td colspan="5">CONTAINS Statement</td></tr>
<tr><td colspan="5">Internal Subprograms or Module</td></tr>
<tr><td colspan="5">END Statement</td></tr>
</table>

The Fortran Character Set

Table 1.1, "Fortran Characters", shows the set of Fortran characters. Character variables and constants can use any ASCII character. The value of the command-line option –Mupcase determines if the compiler distinguishes between case (upper and lower) in identifiers. By default, without the –Mupcase option selected, the compiler does not distinguish between upper and lower case characters in identifiers (upper and lower case are always significant in character constants).

Table 1.1. Fortran Characters

Character	Description	Character	Description
,	Comma	A-Z, a-z	Alphabetic
:	Colon	<space>	Space character
;	Semicolon	=	Equals
_	Underscore character	+	Plus
<	Less than	-	Minus
>	Greater than	*	Asterisk
?	Question mark	/	Slash
%	Percent	(	Left parenthesis
"	Quotation mark	)	Right parenthesis
$	Currency symbol	[	Left bracket
.	Decimal point	]	Right bracket
!	Exclamation mark	<CR>	Carriage return
0-9	Numeric	<TAB>	Tabulation character

Table 1.2, "C Language Character Escape Sequences", shows C language character escape sequences that the PGI Fortran compilers recognize in character string constants when `-Mbackslash` is on the command line. These values depend on the command-line option `-Mbackslash`.

Table 1.2. C Language Character Escape Sequences

Character	Description
\v	vertical tab
\a	alert (bell)
\n	newline
\t	tab
\b	backspace
\f	formfeed
\r	carriage return
\0	null
\'	apostrophe (does not terminate a string)

Character	Description
\"	double quotes (does not terminate a string)
\\	\
\x	x, where x is any other character
\ddd	character with the given octal representation.

Free Form Formatting

Using free form formatting, columns are not significant for the elements of a Fortran line, and a blank or series of blanks or tabs and the context of a token specify the token type. The following rules apply to free form formatting:

- Up to 132 characters are valid per line, and the compiler option –Mextend does not apply.
- A single Fortran line may contain multiple statements, with the ; (semicolon) separating multiple statements on a single line.
- Free format labels are valid at the start of a line.
  - The label must be separated from the remaining statements on the line by at least one blank or a <TAB>.
  - Labels consist of a numeric field drawn from digits 0 to 9.
  - The label cannot be more than 5 characters.
- Either a blank line, or the ! character following a Fortran line indicates a comment. The Fortran text does not contain any of the characters after the !.
- The & character at the end of a line means the following line represents a continuation line.
  - If a continuation line starts with the & character, then the characters following the & are the start of the continuation line.
  - If the continuation line does not start with a &, then all characters on the line are part of the continuation line, including any initial blanks or tabs.

A single Fortran line may contain multiple statements. The ; (semicolon) separates multiple statements on a single line. Free format labels are valid at the start of a line, as long as the label is separated from the remaining statements on the line by at least one blank or a <TAB>. Labels consist of a numeric field drawn from digits 0 to 9. The label cannot be more than 5 characters.

Fixed Formatting

This section describes the two types of fixed formatting that PGI Fortran compilers support: column formatting and tab formatting.

Column Formatting

When using column formatting a Fortran record consists of a sequence of up to 72 or 132 ASCII characters, the last being <CR>. Table 1.3 shows the fixed layout.

Note

For column formatting of 132 characters, you must specify `-Mextend`.

Table 1.3. Fixed Format Record Positions and Fields

Position	Field
1-5	Label field
6	Continuation field
7-72 or 7-132	Statement field

Characters on a line beyond position 72, or position 132 if `-Mextend` is specified, are ignored. In addition, any characters following an exclamation (!) character are considered comments and are thus disregarded during compilation.

Fixed Format Label Field

The label field holds up to five characters. Further, each label must be unique in its program unit.

- The characters C , D, *, or ! in the first character position of a label field indicate a comment line.
- When a numeric field drawn from digits 0 to 9 is placed in the label field, the field is a label.
- A line with no label, and with five space characters or a <TAB> in the label field, is an unlabeled statement.
- Continuation lines must not be labeled.
- A program to only jump to labels that are on executable statements.

Fixed Format Continuation Field

The sixth character position, or the position after the tab, is the continuation field. This field is ignored in comment lines. It is invalid if the label field is not five spaces. A value of 0, <space> or <TAB> indicates the first line of a statement. Any other value indicates a subsequent, continuation line to the preceding statement.

Fixed Format Statement Field

The statement field consists of valid identifiers and symbols, possibly separated by <space> or <TAB> and terminated by <CR>.

Within the statement field, tabs, spaces, comments and any characters found beyond the 72nd character, or position 132 if `-Mextend` is specified, are ignored. As stated earlier, any characters following an exclamation (!) character are considered comments.

Fixed Format Debug Statements

The letter D in column 1 using fixed formatting designates the statement on the specified line is a debugging statement. The compiler treats the debugging statement as a comment, ignoring it, unless the command line option `-Mdlines` is set during compilation. If `-Mdlines` is set, the compiler acts as if the line starting with D were a Fortran statement and compiles the line according to the standard rules.

Tab Formatting

The PGI Fortran compilers support an alternate form of fixed source from called tab source form. A tab formatted source file is made up of a label field, an optional continuation indicator and a statement field. The label field is terminated by a tab character. The label cannot be more than 5 characters.

A continuation line is indicated by a tab character followed immediately by a non-zero digit. The statement field starts after a continuation indicator, when one is present. Again, any characters found beyond the 72nd character, or position 132 if `–Mextend` is specified, are ignored.

Fixed Input File Format Summary

For fixed input file format, the following is true:

- Tab-Format lines are supported.
  - A tab in columns 1-6 ends the statement label field and begins an optional continuation indicator field.
  - If a non-zero digit follows the tab character, the continuation field exists and indicates a continuation field.
  - If anything other than a non-zero digit follows the tab character, the statement body begins with that character and extends to the end of the source statement.

    #### Note

    Note that this does not override Fortran's free source form handling since no valid Fortran statement can begin with a non-zero digit.

  - The tab character is ignored if it occurs in a line except in Hollerith or character constants.
- Input lines may be of varying lengths.
  - If there are fewer than 72 characters, the line is padded with blanks.
  - Characters after the 72nd are ignored unless the `–Mextend` option is used on the command line.

    #### Note

    The `–Mextend` option extends the statement field to position 132.

    When the –Mextend option is used, the input line is padded with blanks if it is fewer than 132 characters; characters after the 132nd are ignored.
- Blank lines are allowed at the end of a program unit.
- The number of continuation lines allowed is extended to 1000 lines.

Include Fortran Source Files

The sequence of consecutive compilation of source statements may be interrupted so that an extra source file can be included. To do this, use the INCLUDE statement which takes the form:

```
INCLUDE "filename"
```

where *filename* is the name of the file to be included. Pairs of either single or double quotes are acceptable enclosing filename.

The INCLUDE file is compiled to replace the INCLUDE statement, and on completion of that source the file is closed and compilation continues with the statement following the INCLUDE.

INCLUDE files are especially recommended when the same COMMON blocks and the same COMMON block data mappings are used in several program units. For example the following statement includes the file MYFILE.DEF.

```
INCLUDE "MYFILE.DEF"
```

Nested includes are allowed, up to a PGI Fortran defined limit of 20.

Recursive includes are not allowed. That is, if a file includes a file, that file may not also include the same file.

Components of Fortran Statements

Fortran program units are made up of statements which consist of expressions and elements. An expression can be broken down to simpler expressions and eventually to its elements combined with operators. Hence the basic building block of a statement is an element.

An element takes one of the following forms:

- A constant represents a fixed value.
- A variable represents a value which may change during program execution.
- An array is a group of values that can be referred to as a whole, as a section, or separately. The separate values are known as the elements of the array. The array has a symbolic name.
- A function reference or subroutine reference is the name of a function or subroutine followed by an argument list. The reference causes the code specified at function/subroutine definition to be executed and if a function, the result is substituted for the function reference.

Symbolic Names

Symbolic names identify different entities in Fortran source code. A symbolic name is a string of letters and digits, which must start with a letter and be terminated by a character not in the symbolic names set (for example a <space> or a <TAB> character). Underscore (_) characters may appear within symbolic names. Only the first 63 characters identify the symbolic name.

Here several examples of symbolic names:

```
NUM
CRA9
numericabcdefghijklmnopqrstuvwxyz
```

The last example is identified by its first 63 characters and is equivalent to:

```
numericabcdefghijklmnopqrstuvwx
```

Some examples of invalid symbolic name include:

```
8Q        Invalid because it begins with a number

FIVE.4      Invalid because it contains a period, an invalid
          character for a symbolic name.
```

Expressions

Each data item, such as a variable or a constant, represents a particular value at any point during program execution. These elements may be combined together to form expressions, using binary or unary operators, so that the expression itself yields a value. A Fortran expression may be any of the following:

- A scalar expression
- An array expression
- A constant expression
- A specification expression
- An initialization expression
- Mixed array and scalar expressions

Forming Expressions

Expressions fall into one of four classes: arithmetic, relational, logical or character, each class described later in this chapter.

An expression is formed like this:

`expr binary-operator expr` or `unary-operator expr`

where `expr` is formed as an expression or as an element.

For example, these are simple expressions whose components are elements. The first expression involves a binary operator and the other two are unary operators.

```
A+B          -C          +D
```

Expression Precedence Rules

Arithmetic, relational and logical expressions may be identified to the compiler by the use of parentheses, as described in "Arithmetic Expressions," on page 9. When no guidance is given to the compiler it imposes a set of precedence rules to identify each expression uniquely. Table 1.1, "Fortran Characters", shows the operator precedence rules for expressions.

Table 1.4. Fortran Operator Precedence

Operator	Evaluated
Unary defined	Highest
**	N/A
* or /	N/A
Unary + or -	N/A

Operator	Evaluated
Binary + or –	N/A
Relational operators: GT., .GE., .LE.	N/A
Relational operators ==, /=	Same precedence
Relational operators <, <=, >, >=	Same precedence
Relational operators .EQ., .NE., .LT.	Same precedence
.NOT.	N/A
.AND.	N/A
.OR.	N/A
.NEQV. and .EQV.	N/A
Binary defined	Lowest

For example, the following two expressions are equivalent. If we set A to 16, B to 4, and C to 2, both expressions equal 8.

```
A/B*C        such as 16 / 4 * 2
```

```
(A/B)*C      such as (16 /4) * 2
```

Another example of equivalent expressions are these:

```
    A*B+B**C     .EQ.   X+Y/Z    .AND.  .NOT. K-3.0 .GT. T
```

```
((((A*B)+(B**C)) .EQ. (X+(Y/Z))) .AND. (.NOT. ((K-3.0) .GT. T)))
```

Arithmetic Expressions

Arithmetic expressions are formed from arithmetic elements and arithmetic operators.

Arithmetic Elements

An arithmetic element may be:

- an arithmetic expression
- a variable
- a constant
- an array element
- a function reference
- a field of a structure

Note

A value should be associated with a variable or array element before it is used in an expression.

Arithmetic Operators

The arithmetic operators specify a computation to be performed on the elements. The result is a numeric result. Table 1.5 shows the arithmetic operators.

Table 1.5. Arithmetic Operators

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition or unary plus
-	Subtraction or unary minus

Arithmetic Operator Precedence

Arithmetic expressions are evaluated in an order determined by a precedence associated with each operator. Table 1.6 shows the precedence of each arithmetic operator.

Table 1.6. Arithmetic Operator Precedence

Operator	Precedence
**	First
* and /	Second
+ and -	Third

This following example is resolved into the arithmetic expressions *(A) + (B * C)* rather than *(A + B) * (C)*.

```
A + B * C
```

Normal ranked precedence may be overcome using parentheses which force the item(s) enclosed to be evaluated first. For example, in the following expression the computer firsts adds A and B, and then multiplies that sum by C.

```
(A + B) * C
```

Arithmetic Expression Types

The type of an arithmetic expression depends on the type of elements in the expression:

INTEGER
: if it contains only integer elements.

REAL
: if it contains only real and integer elements.

DOUBLE PRECISION
: if it contains only double precision, real and integer elements.

COMPLEX
: if any element is complex. Any element which needs conversion to complex will be converted by taking the real part from the original value and setting the imaginary part to zero.

DOUBLE COMPLEX
if any element is double complex.

Table 2.4, "Data Type Ranks" provides more information about these expressions.

Relational Expressions

A relational expression is composed of two arithmetic expressions separated by a relational operator. The value of the expression is true or false (*.TRUE.* or *.FALSE.*) depending on the value of the expressions and the nature of the operator. Table 1.7 shows the relational operators.

Table 1.7. Relational Operators

Operator	Relationship
<	Less than
<=	Less than or equal to
==	Equal to
/=	Not equal to
>	Greater than
>=	Greater than or equal to

In relational expressions the arithmetic elements are evaluated to obtain their values. The relationship is then evaluated to obtain the true or false result. Thus the relational expression:

```
TIME + MEAN .LT. LAST
```

means if the sum of *TIME* and *MEAN* is less than the value of *LAST*, then the result is true, otherwise it is false.

Logical Expressions

A logical expression is composed of two relational or logical expressions separated by a logical operator. Each logical expression yields the value true or false (*.TRUE.* or *.FALSE.*). Table 1.8 shows the logical operators.

Table 1.8. Logical Expression Operators

Operator	Relationship
.AND.	True if both expressions are true.
.OR.	True if either expression or both is true.
.NOT.	This is a unary operator; it is true if the expression is false, otherwise it is false.
.NEQV.	False if both expressions have the same logical value
.XOR.	Same as *.NEQV.*
.EQV.	True if both expressions have the same logical value

In the following example, *TEST* will be *.TRUE.* if *A* is greater than *B* or *I* is not equal to *J+17*.

```
TEST = A .GT. B .OR. I .NE. J+17
```

Character Expressions

An expression of type CHARACTER can consist of one or more printable characters. Its length is the number of characters in the string. Each character is numbered consecutively from left to right beginning with 1. For example:

```
'ab_&'
'A@HJi2'
'var[1,12]'
```

Character Concatenation

A character expression can be formed by concatenating two (or more) valid character expressions using the concatenation operator //. The following table shows several examples of concatenation.

Expression	Value
'ABC'//'YZ'	*"ABCYZ"*
'JOHN '//'SMITH'	*"JOHN SMITH"*
'J '//'JAMES '//'JOY'	*"J JAMES JOY"*

Symbolic Name Scope

Fortran 90/95 scoping is expanded from the traditional FORTRAN 77 capabilities which provide a scoping mechanism using subroutines, main programs, and COMMONs. Fortran 90/95 adds the MODULE statement. Modules provide an expanded alternative to the use of both COMMONs and INCLUDE statements. Modules allow data and functions to be packaged and defined as a unit, incorporating data hiding and using a scope that is determined with the USE statement.

Names of COMMON blocks, SUBROUTINEs and FUNCTIONs are global to those modules that reference them. They must refer to unique objects, not only during compilation, but also in the link stage.

The scope of names other than these is local to the module in which they occur, and any reference to the name in a different module will imply a new local declaration. This includes the arithmetic function statement.

Assignment Statements

A Fortran assignment statement can be any of the following:

- An intrinsic assignment statement
- A statement label assignment
- An array assignment
- A masked array assignment
- A pointer assignment
- A defined assignment

Arithmetic Assignment

The arithmetic assignment statement has the following form:

```
object = arithmetic-expression
```

where *object* is one of the following:

- Variable
- Function name (within a function body)
- Subroutine argument
- Array element
- Field of a structure

The type of *object* determines the type of the assignment (INTEGER, REAL, DOUBLE PRECISION or COMPLEX) and the arithmetic-expression is coerced into the correct type if necessary.

In the case of:

```
complex = real expression
```

the implication is that the real part of the complex number becomes the result of the expression and the imaginary part becomes zero. The same applies if the expression is double precision, except that the expression will be coerced to real.

The following are examples of arithmetic assignment statements.

```
A=(P+Q)*(T/V)
B=R**T**2
```

Logical Assignment

The logical assignment statement has the following form:

```
object = logical-expression
```

where *object* is one of the following:

- Variable
- Function name (only within the body of the function)
- Subroutine argument
- Array element
- A field of a structure

The type of *object* must be logical.

In the following example, *FLAG* takes the logical value *.TRUE.* if *P+Q* is greater than *R*; otherwise *FLAG* has the logical value *.FALSE.*

```
FLAG=(P+Q) .GT. R
```

Character Assignment

The form of a character assignment is:

```
object = character expression
```

where *object* must be of type character, and is one of the following:

- Variable
- Function name (only within the body of the function)
- Subroutine argument
- Array element
- Character substring
- A field of a structure

In addition, these rules apply:

- None of the character positions being defined in *object* can be referenced in the character expression.
- Only such characters as are necessary for the assignment to *object* need to be defined in the character expression.
- The character expression and *object* can have different lengths.
  - When *object* is longer than the character expression, trailing blanks are added to the object.
  - If *object* is shorter than the character expression the right-hand characters of the character expression are truncated as necessary.

In the following example, all the variables and arrays are assumed to be of type character.

```
FILE = 'BOOKS'
PLOT(3:8) = 'PLANTS'
TEXT(I,K+1)(2:B-1) = TITLE//X
```

Listing Controls

The PGI Fortran compilers recognize three compiler directives that affect the program listing process:

%LIST
: Turns on the listing process beginning at the following source code line.

%NOLIST
: Turns off the listing process (including the %NOLIST line itself).

%EJECT
: Causes a new listing page to be started.

Note

These directives have an effect only when the `-Mlist` option is used. All of the directives must begin in column one.

OpenMP Directives

OpenMP directives in a Fortran program provide information that allows the PGF77 and PGF95 compilers to generate executable programs that use multiple threads and processors on a shared-memory parallel (SMP) computer system. An OpenMP directive may have any of the following forms:

```
!$OMP directive
C$OMP directive
*$OMP directive
```

For a complete list and specifications of OpenMP directives supported by the PGF77 and PGF95 compilers, along with descriptions of the related OpenMP runtime library routines, refer to Chapter 8, "*OpenMP Directives for Fortran*," on page 151.

Chapter 2. Fortran Data Types

Every Fortran element and expression has a data type. The data type of an element may be implicit in its definition or explicitly attached to the element in a declaration statement. This chapter describes the Fortran data types and constants that are supported by the PGI Fortran compilers.

Fortran provides two kinds of data types, intrinsic data types and derived data types. Types provided by the language are intrinsic types. Types specified by the programmer and built from the intrinsic data types are called derived types.

Intrinsic Data Types

Fortran provides six different intrinsic data types, listed in Table 2.1, "Fortran Intrinsic Data Types" and Table 2.3, "Data Type Extensions" show variations and different *KIND* of intrinsic data types supported by the PGI Fortran compilers.

Table 2.1. Fortran Intrinsic Data Types

Data Type	Value
INTEGER	An integer number.
REAL	A real number.
DOUBLE PRECISION	A double precision floating point number, real number, taking up two numeric storage units and whose precision is greater than REAL.
LOGICAL	A value which can be either TRUE or FALSE.
COMPLEX	A pair of real numbers used in complex arithmetic. Fortran provides two precisions for COMPLEX numbers.
CHARACTER	A string consisting of one or more printable characters.

Kind Parameter

The Fortran 95 KIND parameter specifies a precision for intrinsic data types. The KIND parameter follows a data type specifier and specifies size or type of the supported data type. A KIND specification overrides the length attribute that the statement implies and assigns a specific length to the item, regardless of the compiler's

command-line options. A KIND is defined for a data type by a PARAMETER statement, using sizes supported on the particular system.

The following are some examples using a KIND specification:

```
INTEGER (SHORT) :: L
REAL (HIGH) B
REAL (KIND=HIGH) XVAR, YVAR
```

These examples require that the programmer use a PARAMETER statement to define kinds:

```
INTEGER, PARAMETER :: SHORT=1
INTEGER HIGH
PARAMETER (HIGH=8)
```

The following table shows several examples of KINDs that a system could support.

Table 2.2. Data Types Kind Parameters

Type	Kind	Size
INTEGER	SHORT	1 byte
INTEGER	LONG	4 bytes
REAL	HIGH	8 bytes

Number of Bytes Specification

The PGI Fortran compilers support a length specifier for some data types. The data type can be followed by a data type length specifier of the form *s, where s is one of the supported lengths for the data type. Such a specification overrides the length attribute that the statement implies and assigns a specific length to the specified item, regardless of the compiler options. For example, REAL*8 is equivalent to DOUBLE PRECISION. Table 2.3 shows the lengths of data types, their meanings, and their sizes.

Table 2.3. Data Type Extensions

Type	Meaning	Size
LOGICAL*1	Small LOGICAL	1 byte
LOGICAL*2	Short LOGICAL	2 bytes
LOGICAL*4	LOGICAL	4 bytes
LOGICAL*8	LOGICAL	8 bytes
BYTE	Small INTEGER	1 byte
INTEGER*1	Same as BYTE	1 byte
INTEGER*2	Short INTEGER	2 bytes
INTEGER*4	INTEGER	4 bytes
INTEGER*8	INTEGER	8 bytes
REAL*4	REAL	4 bytes
REAL*8	DOUBLE PRECISION	8 bytes

Type	Meaning	Size
COMPLEX*8 COMPLEX (Kind=4)	COMPLEX	8 bytes
COMPLEX*16 COMPLEX (Kind=8)	DOUBLE COMPLEX	16 bytes

The BYTE type is treated as a signed one-byte integer and is equivalent to INTEGER*1.

Note

Assigning a value too big for the data type to which it is assigned is an undefined operation.

A specifier is allowed after a CHARACTER function name even if the CHARACTER type word has a specifier. In the following example, the function size specification C*8 overrides the CHARACTER*4 specification.

```
CHARACTER*4 FUNCTION C*8 (VAR1)
```

Logical data items can be used with any operation where a similar sized integer data item is permissible and vice versa. The logical data item is treated as an integer or the integer data item is treated as a logical of the same size and no type conversion is performed.

Floating point data items of type REAL or DOUBLE PRECISION may be used as array subscripts, in computed GOTOs, in array bounds and in alternate returns. The floating point data item is converted to an integer.

The data type of the result of an arithmetic expression corresponds to the type of its data. The type of an expression is determined by the rank of its elements. Table 2.4 shows the ranks of the various data types, from lowest to highest.

Note

A variable of logical type may appear in an arithmetic context, and the logical type is then treated as an integer of the same size.

Table 2.4. Data Type Ranks

Data Type	Rank
LOGICAL	1 (lowest)
LOGICAL*8	2
INTEGER*2	3
INTEGER*4	4
INTEGER*8	5
REAL*4	6
REAL*8 (Double precision)	7
COMPLEX*8 (Complex)	8

Data Type	Rank
COMPLEX*16 (Double complex)	9 (highest)

The data type of a value produced by an operation on two arithmetic elements of different data types is the data type of the highest-ranked element in the operation. The exception to this rule is that an operation involving a COMPLEX*8 element and a REAL*8 element produces a COMPLEX*16 result. In this operation, the COMPLEX*8 element is converted to a COMPLEX*16 element, which consists of two REAL*8 elements, before the operation is performed.

In most cases, a logical expression will have a LOGICAL*4 result. In cases where the hardware supports LOGICAL*8 and if the expression is LOGICAL*8, the result may be LOGICAL*8.

Constants

A constant is an unchanging value that can be determined at compile time. It takes a form corresponding to one of the data types. The PGI Fortran compilers support decimal (INTEGER and REAL), unsigned binary, octal, hexadecimal, character and Hollerith constants.

The use of character constants in a numeric context, for example, in the right-hand side of an arithmetic assignment statement, is supported. These constants assume a data type that conforms to the context in which they appear.

Integer Constants

The form of a decimal integer constant is:

```
[s]d1d2...dn [ _ kind-parameter ]
```

where `s` is an optional sign and `di` is a digit in the range 0 to 9. The optional _kind#parameter is a Fortran 90/95 feature supported only by PGF95, and specifies a supported kind. The value of an integer constant must be within the range for the specified kind.

The value of an integer constant must be within the range -2147483648 (-2^{31}) to 2147483647 ($2^{31} - 1$) inclusive. Integer constants assume a data type of INTEGER*4 and have a 32-bit storage requirement.

The `-i8` compilation option causes all data of type INTEGER to be promoted to an 8 byte INTEGER. The –i8 option does not override an explicit data type extension size specifier, such as INTEGER*4. The range, data type and storage requirement change if the –i8 flag is specified, although this flag is not supported on all x86 targets. With the –i8 flag, the range for integer constants is -2^{63} to ($2^{63} - 1$)), and in this case the value of an integer constant must be within the range -9223372036854775808 to 9223372036854775807. If the constant does not fit in an INTEGER*4 range, the data type is INTEGER*8 and the storage requirement is 64 bits.

Here are several examples of integer constants:

```
+2
-36
437
-36_SHORT
369_I2
```

Binary, Octal and Hexadecimal Constants

The PGI compilers and Fortran 90/95 support various types of constants in addition to decimal constants. Fortran allows unsigned binary, octal, or hexadecimal constants in DATA statements. PGI compilers support these constants in DATA statements, and additionally, support some of these constants outside of DATA statements. For more information on support of these constants, refer to "Fortran Binary, Octal and Hexadecimal Constants," on page 28.

Real Constants

Real constants have two forms, scaled and unscaled. An unscaled real constant consists of a signed or unsigned decimal number (a number with a decimal point). A scaled real constant takes the same form as an unscaled constant, but is followed by an exponent scaling factor of the form:

```
E+digits [_ kind-parameter ]
Edigit [_ kind-parameter ]
E-digits [_ kind-parameter ]
```

where digits is the scaling factor, the power of ten, to be applied to the unscaled constant. The first two forms above are equivalent, that is, a scaling factor without a sign is assumed to be positive. Table 2.5 shows several real constants.

Table 2.5. Examples of Real Constants

Constant	Value
1.0	unscaled single precision constant
1.	unscaled single precision constant
-.003	signed unscaled single precision constant
-.003_LOW	signed unscaled constant with kind LOW
-1.0	signed unscaled single precision constant
6.1E2_LOW	is equivalent to 610.0 with kind LOW
+2.3E3_HIGH	is equivalent to 2300.0 with kind HIGH
6.1E2	is equivalent to 610.0
+2.3E3	is equivalent to 2300.0
-3.5E-1	is equivalent to -0.35

Double Precision Constants

A double precision constant has the same form as a scaled REAL constant except that the E is replaced by D and the kind parameter is not permitted. For example:

```
D+digits
Ddigit
D-digits
```

Table 2.6 shows several double precision constants.

Table 2.6. Examples of Double Precision Constants

Constant	Value
6.1D2	is equivalent to 610.0
+2.3D3	is equivalent to 2300.0
-3.5D-1	is equivalent to -0.35
+4D4	is equivalent to 40000

Complex Constants

A complex constant is held as two real or integer constants separated by a comma and surrounded by parentheses. The first real number is the real part and the second real number is the imaginary part. Together these values represent a complex number. Integer values supplied as parameters for a COMPLEX constant are converted to REAL numbers. Here are several examples:

```
(18,-4)
(3.5,-3.5)
(6.1E2,+2.3E3)
```

Double Complex Constants

A complex constant is held as two double constants separated by a comma and surrounded by parentheses. The first double is the real part and the second double is the imaginary part. Together these values represent a complex number. Here is an example:

```
(6.1D2,+2.3D3)
```

Logical Constants

A logical constant is one of:

```
.TRUE. [ _ kind-parameter ]
.FALSE.[ _ kind-parameter ]
```

The logical constants .TRUE. and .FALSE. are by default defined to be the four-byte values -1 and 0 respectively. A logical expression is defined to be .TRUE. if its least significant bit is 1 and .FALSE. otherwise[1].

Here are several examples:

```
.TRUE.
.FALSE.
.TRUE._BIT
```

The abbreviations .T. and .F. can be used in place of .TRUE. and .FALSE. in data initialization statements and in NAMELIST input.

Character Constants

Character string constants may be delimited using either an apostrophe (') or a double quote ("). The apostrophe or double quote acts as a delimiter and is not part of the character constant. Use double quotes

[1]The option –Munixlogical defines a logical expression to be TRUE if its value is non-zero, and FALSE otherwise; also, the internal value of .TRUE. is set to one. This option is not available on all target systems.

or two apostrophes together to include an apostrophe as part of an expression. If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote or backslash escape. Within character constants, blanks are significant. For further information on the use of the backslash character, refer to `-Mbackslash` information in the User's Guide.

A character constant is one of:

```
[ kind-parameter_ ] "[characters]"
[ kind-parameter_ ] '[characters]'
```

Here are several examples of character constants.

```
'abc'
'abc '
'ab''c'
"Test Word"
GREEK_"μ"
```

A zero length character constant is written as '' or "".

If a character constant is used in a numeric context, for example as the expression on the right side of an arithmetic assignment statement, it is treated as a Hollerith constant. The rules for typing and sizing character constants used in a numeric context are described in "Hollerith Constants," on page 30.

Parameter Constants

The PARAMETER statement permits named constants to be defined. For more details on defining constants, refer to the description of the PARAMETER statement in Chapter 3, "*Fortran Statements*".

Structure Constructors

A structure constructor looks like a function call. It is a mechanism whose purpose is to specify a value of a derived type or of a type alias that describes a derived type. The constructor specifies 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.

Syntax

A structure constructor is the name of the type followed by a sequence of component values in parentheses. The format for a structure_constructor is one of the following:

```
type_name (expr_list)
```

```
type_alias_name (expr_list)
```

In Fortran 2003, there are three significant enhancements to structure constructors that make structure constructors more like built-in generic functions that can be overridden when necessary.

- Component names can be used as keywords, the same way that dummy argument names can be used as argument keywords

- Values can be omitted for components that have default initialization.
- Type names can be the same as generic function names; references are resolved by choosing a suitable function (if the syntax matches the function's argument list) and treating as a structure constructor only if no function matches the actual arguments

The following rules apply to structure constructors:

- A structure constructor must not appear before that type is defined.
- There must be a value in the expression list for each component unless that component has default initialization.
- The expressions must agree in number and order with the components of the derived type. Values may be converted to agree in type, kind, length, and, in some cases, rank, with the components.
- 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.
- If a component is an explicit-shape array, such as a nonpointer array or a nonallocatable 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 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.
- If a component has the ALLOCATABLE attribute, its value in the expression list must have the same rank if it is an array or must be scalar if it is scalar. The value must be one of the following:
  - A call to the NULL(3i) intrinsic command without any arguments. The allocatable component receives a ‘‘not currently allocated’’ status.
  - A variable that has the ALLOCATABLE attribute. The allocatable component receives the variable's allocation status and, if allocated, shape and value.
  - An expression. The allocatable component receives the ‘‘currently allocated’’ status and the same value and shape of the expression.

Derived Types

Unlike the intrinsic types that are defined by the language, you must define derived types. A *derived type* is a type made up of components whose type is either intrinsic or another derived type. These types have the same functionality 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 either an intrinsic or derived type.

The TYPE and END TYPE keywords define a derived type. The definition of a variable of the new type is called a TYPE statement.

Syntax

For derived type definition:

```
derived_type_stmt
      [data_component_part]
end_type_stmt
```

For a derived type statement:

```
TYPE [ [, type_attr_spec_list ] :: ] type_name
```

For example, the following derived type declaration defines the type PERSON and the array CUSTOMER of type PERSON:

```
! Declare a structure to define a person derived type
TYPE PERSON
 INTEGER ID
 LOGICAL LIVING
 CHARACTER(LEN=20) FIRST, LAST, MIDDLE
 INTEGER AGE
END TYPE PERSON
TYPE (PERSON) CUSTOMER(10)
```

A *derived type statement* consists of the statements between the TYPE and END TYPE statements. In the previous example, the derived-type statement for PERSON consists of all the statements between TYPE PERSON and END TYPE PERSON.

Notice in this example that CUSTOMER is a variable of type PERSoN. Use of parentheses in the TYPE statement indicate a reference to the derived type PERSON rather than declaration of a derived type.

The % character accesses the components of a derived type. For example, to assign the value 12345 as the ID of the first customer, you might use the following statement:

```
CUSTOMER(1)%ID = 12345
```

Deferred Type Parameters

A *deferred type parameter* is a type parameter that has no defined value until it is given one. In Fortran 2003, deferred type parameters are available both for CHARACTER length and for parameterized derived types.

A variable with a deferred type parameter must have the ALLOCATABLE or POINTER attribute. The value of a deferred type parameter depends on this attribute:

- For an allocatable variable, the value of a deferred type parameter is determined by allocation - either by a typed allocation, or by an intrinsic assignment with automatic reallocation.
- For a pointer, the value of a deferred type parameter is the value of the type parameter of its target.

Typed Allocation

A length type parameter that is deferred has no defined value until it is given one by the ALLOCATE statement or by point assignment. There are a couple rules that apply with typed allocation and deferred type parameters:

- If the length parameters of an item being allocated is assumed, it must be specified as an asterisk (*) in the *type-spec* of the ALLOCATE statement.
- Since there can only be one type-spec in an ALLOCATE statement, it must be suitable for all the items being allocated. For example, if any of the allocatable items is a dummy argument, then they must all be dummy arguments.

Arrays

Arrays in Fortran are not data types, but are data objects of intrinsic or derived type with special characteristics. A dimension statement provides a data type with one or more dimensions. There are several differences between Fortran 90/95 and traditional FORTRAN 77 arrays.

> **Note**
>
> Fortran 90/95 supports all FORTRAN 77 array semantics.

An array is a group of consecutive, contiguous storage locations associated with a symbolic name which is the array name. Each individual element of storage, called the array element, is referenced by the array name modified by a list of subscripts. Arrays are declared with type declaration statements, DIMENSION statements and COMMON statements; they are not defined by implicit reference. These declarations will introduce an array name and establish the number of dimensions and the bounds and size of each dimension. If a symbol, modified by a list of subscripts is not defined as an array, then it will be assumed to be a FUNCTION reference with an argument list.

Fortran 90/95 arrays are "objects" and operations and expressions involving arrays may apply to every element of the array in an unspecified order. For example, in the following code, where A and B are arrays of the same shape (conformable arrays), the following expression adds six to every element of B and assigns the results to the corresponding elements of A:

```
A = B + 6
```

Fortran arrays may be passed with unspecified shapes to subroutines and functions, and sections of arrays may be used and passed as well. Arrays of derived type are also valid. In addition, allocatable arrays may be created with deferred shapes (number of dimensions is specified at declaration, but the actual bounds and size of each dimension are determined when the array is allocated while the program is running).

Array Declaration Element

An array declaration has the following form:

```
name([lb:]ub[,[lb:]ub]...)
```

where name is the symbolic name of the array, `lb` is the specification of the lower bound of the dimension and `ub` is the specification of the upper bound. The upper bound, `ub` must be greater than or equal to the lower bound lb. The values `lb` and `ub` may be negative. The bound `lb` is taken to be 1 if it is not specified. The difference (`ub-lb+1`) specifies the number of elements in that dimension. The number of `lb,ub` pairs specifies the rank of the array. Assuming the array is of a data type that requires N bytes per element, the total amount of storage of the array is:

```
N*(ub-lb+1)*(ub-lb+1)*...
```

The dimension specifiers of an array subroutine argument may themselves be subroutine arguments or members of COMMON.

Deferred Shape Arrays

Deferred-shape arrays are those arrays whose shape can be changed by an executable statement. Deferred-shape arrays are declared with a rank, but with no bounds information. They assume their shape when either an ALLOCATE statement or a REDIMENSION statement is encountered.

For example, the following statement declares a deferred shape REAL array A of rank two:

```
REAL A(:, :)
```

Subscripts

A subscript is used to specify an array element for access. An array name qualified by a subscript list has the following form:

```
name(sub[,sub]...)
```

where there must be one sub entry for each dimension in array name.

Each sub must be an integer expression yielding a value which is within the range of the lower and upper bounds. Arrays are stored as a linear sequence of values in memory and are held such that the first element is in the first store location and the last element is in the last store location. In a multi-dimensional array the first subscript varies more rapidly than the second, the second more rapidly than the third, and so on (column major order).

Character Substring

A character substring is a contiguous portion of a character variable and is of type character. A character substring can be referenced, assigned values and named. It can take either of the following forms:

```
character_variable_name(x1:x2)
character_array_name(subscripts)(x1:x2)
```

where x1 and x2 are integers and x1 denotes the left-hand character position and x2 the right-hand one. These are known as substring expressions. In substring expressions x1 must be both greater than or equal to 1 and less than x2 and x2 must be less than or equal to the length of the character variable or array element.

For example, the following expression indicates the characters in positions 2 to 4 of character variable *J*.

```
J(2:4)
```

This next expression indicates characters in positions 1 to 4 of array element *K(3,5)*.

```
K(3,5)(1:4)
```

A substring expression can be any valid integer expression and may contain array elements or function references.

Array Constructor Syntax

In Fortran 2003, array constructors may be bracketed with [ ] instead of (/ /). In addition, array constructors may contain a type specification that explicitedly specifies the type and type parameters of the array. These

constructors begin with a type specification followed by a double colon (::), as illustrated in the examples later in this section. The general format for this type specification is this:

```
(/ type-spec :: ac-value-list /)
```

Note

If the `type-spec` is absent in the array specification, Fortran 95 rules apply; and all items must have the same type and type parameters.

The type-spec syntax is useful for a number of reasons, such as these:

- It simplifies zero-sized constructors.
- It provides assignment conversions that eliminate the need for users to pad all characters in an array to the same length.
- It makes some constructors easiers, such as allowing users to specify either real or integer values in a complex array.

Examples

```
[ character(len=12) : :  ‘crimson’, ‘cream’, ‘purple’, ‘gold’ ]

[ complex(kind(0d0) ;; 1, (0,1), 3.3333d0 ]

[ matrix(kind=kind(0,0), n=5, m=7) : ]   !zero-sized array

[ Logical :: ]                    ! Zero-sized logical array

[ Double Precision :: 17.5, 0, 0.1d0 ]     ! Conversions
```

Fortran Pointers and Targets

Fortran pointers are similar to allocatable arrays. Pointers are declared with a type and a rank; they do not actually represent a value, however, but represent a value's address. Fortran 90/95 has a specific assignment operator, =>, for use in pointer assignments.

Fortran Binary, Octal and Hexadecimal Constants

The PGI Fortran compilers support two representations for binary, octal, and hexadecimal numbers: the standard Fortran 90/95 representation and the PGI extension representation. In addition, PGI supports an alternate representation, described in the next section.

Fortran supports binary, octal and hexadecimal constants in DATA statements.

Binary Constants

The form of a binary constant is:

```
B'b1b2...bn'
B"b1b2...bn"
```

where `bi` is either 0 or 1., such as B’01001001’

Octal Constants

The form of an octal constant is:

```
O'c1c2...cn'
O"c1c2...cn"
```

where `ci` is in the range 0 through 7. such as O'043672'

Hexadecimal Constants

The form of a hexadecimal constant is:

```
Z'a1a2...an'
Z"a1a2...an"
```

where `ai` is in the range 0 through 9 or a letter in the range A through F or a through f (case mixing is allowed), such as Z'8473Abc58' or "BF40289cd"X .

Octal and Hexadecimal Constants - Alternate Forms

The PGF95 compiler supports additional extensions. This is an alternate form for octal constants, outside of DATA statements. The form for an octal constant is:

```
'c1c2...cn'O
```

where `ci` is a digit in the range 0 to 7.

The form of a hexadecimal constant is:

```
'a1a2...an'X
"a1a2...an"X
```

where `ai` is a digit in the range 0 to 9 or a letter in the range A to F or a to f (case mixing is allowed). Up to 64 bits (22 octal digits or 16 hexadecimal digits) can be specified.

Octal and hexadecimal constants are stored as either 32-bit or 64-bit quantities. They are padded on the left with zeroes if needed and assume data types based on how they are used.

The following are the rules for converting these data types:

- A constant is always either 32 or 64 bits in size and is typeless. Sign-extension and type-conversion are never performed. All binary operations are performed on 32-bit or 64-bit quantities. This implies that the rules to follow are only concerned with mixing 32-bit and 64-bit data.
- When a constant is used with an arithmetic binary operator (including the assignment operator) and the other operand is typed, the constant assumes the type and size of the other operand.
- When a constant is used in a relational expression such as .EQ., its size is chosen from the operand having the largest size. This implies that 64-bit comparisons are possible.
- When a constant is used as an argument to the generic AND, OR, EQV, NEQV, SHIFT, or COMPL function, a 32-bit operation is performed if no argument is more than 32 bits in size; otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.
- When a constant is used as an actual argument in any other context, no data type is assumed; however, a length of four bytes is always used. If necessary, truncation on the left occurs.

- When a specific 32-bit or 64-bit data type is required, that type is assumed for the constant. Array subscripting is an example.
- When a constant is used in a context other than those mentioned above, an INTEGER*4 data type is assumed. Logical expressions and binary arithmetic operations with other untyped constants are examples.
- When the required data type for a constant implies that the length needed is more than the number of digits specified, the leftmost digits have a value of zero. When the required data type for a constant implies that the length needed is less than the number of digits specified, the constant is truncated on the left. Truncation of nonzero digits is allowed.

In the following example, the constant I (of type INTEGER*4) and the constant J (of type INTEGER*2) are assigned hex values 1234 and 4567, respectively. The variable D (of type REAL*8) has the hex value x4000012345678954 after its second assignment:

```
I = '1234'X ! Leftmost Pad with zero
J = '1234567'X ! Truncate Leftmost 3 hex digits
D = dble('40000123456789ab'X)
D = NEQV(D,'ff'X) ! 64-bit Exclusive Or
```

Hollerith Constants

The form of a Hollerith constant is:

```
nHc1c2...cn
```

where *n* specifies the positive number of characters in the constant and cannot exceed 2000 characters.

A Hollerith constant is stored as a byte string with four characters per 32-bit word. Hollerith constants are untyped arrays of INTEGER*4. The last word of the array is padded on the right with blanks if necessary. Hollerith constants cannot assume a character data type and cannot be used where a character value is expected.

The data type of a Hollerith constant used in a numeric expression is determined by the following rules:

- Sign-extension is never performed.
- The byte size of the Hollerith constant is determined by its context and is not strictly limited to 32 or 64 bits like hexadecimal and octal constants.
- When the constant is used with a binary operator (including the assignment operator), the data type of the constant assumes the data type of the other operand.
- When a specific data type is required, that type is assumed for the constant. When an integer or logical is required, INTEGER*4 and LOGICAL*4 are assumed. When a float is required, REAL*4 is assumed (array subscripting is an example of the use of a required data type).
- When a constant is used as an argument to certain generic functions (AND, OR, EQV, NEQV, SHIFT, and COMPL), a 32-bit operation is performed if no argument is larger than 32 bits; otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.
- When a constant is used as an actual argument, no data type is assumed and the argument is passed as an INTEGER*4 array. Character constants are passed by descriptor only.
- When a constant is used in any other context, a 32-bit INTEGER*4 array type is assumed.

When the length of the Hollerith constant is less than the length implied by the data type, spaces are appended to the constant on the right. When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right.

Structures

A structure, a DEC extension to FORTRAN 77, is a user-defined aggregate data type having the following form:

```
STRUCTURE [/structure_name/][field_namelist]
 field_declaration
 [field_declaration]
 ...
 [field_declaration]
END STRUCTURE
```

Where:

structure_name
: is unique and is used both to identify the structure and to allow its use in subsequent RECORD statements.

field_namelist
: is a list of fields having the structure of the associated structure declaration. A field_namelist is allowed only in nested structure declarations.

field_declaration
: can consist of any combination of substructure declarations, typed data declarations, union declarations or unnamed field declarations.

The following rules apply:

- Field names within the same declaration nesting level must be unique.
- An inner structure declaration can include field names used in an outer structure declaration without conflict.
- Records use periods to separate fields, so it is not legal to use relational operators (for example, .EQ., .XOR.), logical constants (.TRUE. or .FALSE.), or logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.
- Fields within structures conform to machine-dependent alignment requirements, that is, fields in a structure are aligned as required by hardware.
  - A structure's storage requirements are machine-dependent.
  - Alignment of fields provides a C-like "struct" building capability and allows convenient inter-language communications.
- Because explicit padding of records is not necessary, the compiler recognizes the %FILL intrinsic, but performs no action in response to it.
- Data initialization can occur for the individual fields.

Records

A record, a DEC extension to FORTRAN 77, is a user-defined aggregate data item having the following form:

```
RECORD /structure_name/record_namelist
 [,/structure_name/record_namelist]
 ...
 [,/structure_name/record_namelist]
```

where:

structure_name
: is the name of a previously declared structure.

record_namelist
: is a list of one or more variable or array names separated by commas.

You create memory storage for a record by specifying a structure name in the RECORD statement. You define the field values in a record either by defining them in the structure declaration or by assigning them with executable code.

You can access individual fields in a record by combining the parent record name, a period (.), and the field name (for example, recordname.fieldname). For records, a scalar reference means a reference to a name that resolves to a single typed data item (for example, INTEGER), while an aggregate reference means a reference that resolves to a structured data item.

Scalar field references may appear wherever normal variable or array elements may appear with the exception of COMMON, SAVE, NAMELIST, DATA and EQUIVALENCE statements. Aggregate references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms.

The following example shows RECORD and STRUCTURE usage.

```
STRUCTURE /person/
! Declare a structure defining a person
! Person has id, names, age, and may or not be living
 INTEGER id
 LOGICAL living
 CHARACTER*5 first, last, middle
 INTEGER age
END STRUCTURE

 ! Define population to be an array where each element is of
 ! type person. Also define a variable, me, of type person.
RECORD /person/ population(2), me
 ...
me.age = 34          ! Assign values for the variable me
me.living = .TRUE.   ! to some of the fields.
me.first = 'Steve'
me.id = 542124822
 ...
population(1).last = 'Jones'   ! Assign the "last" field of
                               ! element 1 of array population.
population(2) = me       ! Assign all values of record
                         ! "me" to the record population(2)
...
```

UNION and MAP Declarations

A UNION declaration, a DEC extension to FORTRAN 77, is a multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. It declares groups of fields that share a common location within a structure.

Declaring and Defining Fields

Each group of fields within a UNION declaration is declared by a MAP declaration, with one or more fields per MAP declaration.

You use union declarations when you want to use the same area of memory to alternately contain two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in a program, that field and any other fields in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration.

Format

The format of a UNION statement is illustrated in the following example:

```
UNION
 map_declaration
 [map_declaration]
 ...
 [map_declaration]
END UNION
```

The format of the map_declaration is as follows:

```
MAP
 field_declaration
 [field_declaration]
 ...
 [field_declaration]
END MAP
```

where *field_declaration* is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union.

With respect to UNION and MAP statements, the following is true:

- Data can be initialized in field declaration statements in union declarations.

  ### Note

  It is illegal to initialize multiple map declarations in a single union.

- Field alignment within multiple map declarations is performed as previously defined in structure declarations.
- The size of the shared area for a union declaration is the size of the largest map defined for that union.
- The size of a map is the sum of the sizes of the field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to what happens using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the C language requires one to associate a name with each "map" (union). Fortran field names must be unique within the same declaration nesting level of maps.

The following example shows RECORD, STRUCTURE, MAP and UNION usage. The size of each element of the recarr array would be the size of typetag (4 bytes) plus the size of the largest MAP, in this case, the employee map (24 bytes).

```
STRUCTURE /account/
 INTEGER typetag        ! Tag to determine defined map.
 UNION
 MAP                    ! Structure for an employee
 CHARACTER*12 ssn          ! Social Security Number
 REAL*4 salary             ! Salary
 CHARACTER*8 empdate       ! Employment date
 END MAP

 MAP                    ! Structure for a customer
 INTEGER*4 acct_cust        ! 4-digit account
 REAL*4 credit_amt          ! credit amount
 CHARACTER*8 due_date       ! due date
 END MAP

 MAP                   ! Structure for a supplier
 INTEGER*4 acct_supp        ! supply account
 REAL*4 debit_amt           ! debit amount
 BYTE num_items             ! number of items
 BYTE items(12)             ! items supplied
 END MAP

 END UNION
END STRUCTURE
RECORD /account/ recarr(1000)
```

Data Initialization

Data initialization is allowed within data type declaration statements. This is an extension to the Fortran language. Data is initialized by placing values bounded by slashes immediately following the symbolic name (variable or array) to be initialized. Initialization of fields within structure declarations is allowed, but initialization of unnamed fields and records is not.

Hollerith, octal and hexadecimal constants can be used to initialize data in both data type declarations and in DATA statements. Truncation and padding occur for constants that differ in size from the declared data item (as specified in the discussion of constants).

Pointer Variables

The POINTER statement, a CRAY extension to FORTRAN 77 which is distinct from the Fortran 90/95 POINTER specification statement or attribute, declares a scalar variable to be a pointer variable of data type INTEGER, and another variable to be its pointer-based variable.

The syntax of the POINTER statement is:

```
POINTER (p1, v1) [, (p2, v2) ...]
```

v1 and v2
: are pointer-based variables. A pointer-based variable can be of any type, including STRUCTURE. A pointer-based variable can be dimensioned in a separate type, in a DIMENSION statement, or in the POINTER statement. The dimension expression may be adjustable, where the rules for adjustable dummy arrays regarding any variables which appear in the dimension declarators apply.

p1 and p2
: are the pointer variables corresponding to v1 and v2. A pointer variable may not be an array. The pointer is an integer variable containing the address of a pointer-based variable. The storage located by the pointer variable is defined by the pointer-based variable (for example, array, data type, etc.). A reference to a pointer-based variable appears in Fortran statements like a normal variable reference (for example, a local variable, a COMMON block variable, or a dummy variable). When the based variable is referenced, the address to which it refers is always taken from its associated pointer (that is, its pointer variable is dereferenced).

The pointer-based variable does not have an address until its corresponding pointer is defined.

The pointer is defined in one of the following ways:

- By assigning the value of the LOC function.
- By assigning a value defined in terms of another pointer variable.
- By dynamically allocating a memory area for the based variable. If a pointer-based variable is dynamically allocated, it may also be freed.

The following code illustrates the use of pointers:

```
REAL XC(10)
COMMON IC, XC
POINTER (P, I)
POINTER (Q, X(5))
P = LOC(IC)
I = 0 ! IC gets 0
P = LOC(XC)
Q = P + 20 ! same as LOC(XC(6))
X(1) = 0 ! XC(6) gets 0
ALLOCATE (X) ! Q locates an allocated memory area
```

Restrictions

The following restrictions apply to the POINTER statement:

- No storage is allocated when a pointer-based variable is declared.
- If a pointer-based variable is referenced, its pointer variable is assumed to be defined.
- A pointer-based variable may not appear in the argument list of a SUBROUTINE or FUNCTION and may not appear in COMMON, EQUIVALENCE, DATA, NAMELIST, or SAVE statements.
- A pointer-based variable can be adjusted only in a SUBROUTINE or FUNCTION subprogram.

  If a pointer-based variable is an adjustable array, it is assumed that the variables in the dimension declarators are defined with an integer value at the time the SUBROUTINE or FUNCTION is called.

For a variable which appears in a pointer-based variable's adjustable declarator, modifying its value during the execution of the SUBROUTINE or FUNCTION does not modify the bounds of the dimensions of the pointer-based array.

- A pointer-based variable is assumed not to overlap with another pointer-based variable.

Pointer Assignment

Fortran 2003 extends pointer assignment for arrays allowing lower bounds and possibly upper bounds to be specified.

Syntax:

```
p(0:,0:) => a
```

The lower bounds may be any scalar integer expressions when upper bounds are specified. Further, remapping of the elements of a target array is permitted, as shown in this example:

```
p(1:m,1:2*m) => a(1:2*m)
```

Description

The following is true for pointer assignments involving arrays:

- The bounds may be any scalar integer expressions.
- The assignment is in array-element order and the target array must be large enough.
- When remapping occurs, the target must be rank-one; otherwise, the ranks of the pointer and target must be the same.

  ```
  a => b(1:10:2)
  ```

- Length type parameters of the pointer may be deferred, that is, declared with a colon.
- Pointer assignment gives these the values of the corresponding parameters of the target.
- All other type parameters of the pointer must have the same values as the corresponding type parameters of the target.

Chapter 3. Fortran Statements

This chapter describes each of the Fortran statements supported by the PGI Fortran compilers. Each description includes a brief summary of the statement, a syntax description, a complete description and an example. The statements are listed in alphabetical order. The first section lists terms that are used throughout the chapter.

Statement Format Overview

This section lists terms that are used throughout the chapter and provides information on how to interpret the information in the statement descriptions. This chapter only provides detailed descriptions for statements that are extensions of the standard Fortran language definitions. For details on the standard statements, refer to the Fortran language specifications readily available on the internet. The `Origin` column in the tables in this chapter provides the Fortran language origin of the statement; for example F95 indicates the statement is from Fortran 95.

Definition of Statement-related Terms

character scalar memory reference
: is a character variable, a character array element, or a character member of a structure or derived type.

integer scalar memory reference
: is an integer variable, an integer array element, or an integer member of a structure or derived type.

logical scalar memory reference
: is a logical variable, a logical array element, or a logical member of a structure or derived type.

obsolescent
: The statement is unchanged from the FORTRAN 77 definition but has a better replacement in Fortran 95.

Origin of Statement

At the top of each reference page is a brief description of the statement followed by a header that indicates the origin of the statement. The following list describes the meaning of the origin header.

F77
: FORTRAN 77 statements that are essentially unchanged from the original FORTRAN 77 standard and are supported by the PGF77 compiler.

F77 extension
: The statement is an extension to the Fortran language.

F90/F95
: The statement is either new for Fortran 90/95 or significantly changed in Fortran 95 from its original FORTRAN 77 definition and is supported by the PGF95 compiler.

F2003
: The statement is new for Fortran 2003.

CMF

List-related Notation

Several statements allow lists of a specific type of data. For example, the ALLOCATABLE statement allows a list in which each element of a deferred-array-spec. The notation used in statements is this:

- Within the statement, the notation is `foo-list`, such as `deferred-array-spec-list`.
- When the list elements have a specific format that is defined, the reference is just to that element, such as `deferred-array-spec`.

As in Fortran, the list is a set of comma-separated values.

Fortran Statement Summary Table

This section contains an alphabetical listing with a brief one-line description of the Fortran statements that PGI supports.Later in this chapter there is more detailed description of the statements that are extensions to the standard Fortran definitions.

Table 3.1. Statement Summary Table

Statement	Origin	Description
ACCEPT	F77	Causes formatted input to be read on standard input.
ALLOCATABLE	F90	Specifies that an array with fixed rank but deferred shape is available for a future ALLOCATE statement.
ALLOCATE	F90	Allocates storage for each allocatable array, pointer object, or pointer-based variable that appears in the statements; declares storage for deferred-shape arrays.
ARRAY	CMF	Defines the number of dimensions in an array that may be defined, and the number of elements and bounds in each dimension. [Not in PVF]
ASSIGN	F77	[Obsolescent]. Assigns a statement label to a variable.
ASSOCIATE	F2003	Associates a name either with a variable or with the value of an expression for the duration of a block.
ASYNCHRONOUS	F77	Warns the compiler that incorrect results might occur for optimizations involving movement of code across wait statements or statements that cause wait operations.

Statement	Origin	Description
BACKSPACE	F77	Positions the file connected to the specified unit to before the preceding record.
BLOCK DATA	F77	Introduces a number of non-executable statements that initialize data values in COMMON tables
BYTE	F77 ext	Establishes the data type of a variable by explicitly attaching the name of a variable to a 1-byte integer, overriding implied data typing.
CALL	F77	Transfers control to a subroutine.
CASE	F90	Begins a case-statement-block portion of a SELECT CASE statement.
CHARACTER	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a character data type, overriding the implied data typing.
CLOSE	F77	Terminates the connection of the specified file to a unit.
COMMON	F77	Defines global blocks of storage that are either sequential or non-sequential; can be either a static or dynamic form.
COMPLEX	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a complex data type, overriding implied data typing.
CONTAINS	F90 F2003	Precedes a subprogram, a function or subroutine and indicates the presence of the subroutine or function definition inside a main program, external subprogram, or module subprogram. In F2003 a `contains` statement can also appear in a derived type right before any type bound procedure definitions.
CONTINUE	F77	Passes control to the next statement.
CYCLE	F90	Interrupts a DO construct execution and continues with the next iteration of the loop.
DATA	F77	Assigns initial values to variables before execution.
DEALLOCATE	F77	Causes the memory allocated for each pointer-based variable or allocatable array that appears in the statement to be deallocated (freed); also deallocates storage for deferred-shape arrays.
DECODE	F77 ext	Transfers data between variables or arrays in internal storage and translates that data from character form to internal form, according to format specifiers.
DIMENSION	F90	Defines the number of dimensions in an array and the number of elements in each dimension.
DO (Iterative)	F90	Introduces an iterative loop and specifies the loop control index and parameters.
DO WHILE	F77	Introduces a logical do loop and specifies the loop control expression.
DOUBLE COMPLEX	F77	Establishes the data type of a variable by explicitly attaching the name of a variable to a double complex data type, overriding implied data typing.

Statement	Origin	Description
DOUBLE PRECISION	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a double precision data type, overriding implied data typing.
ELSE	F77	Begins an ELSE block of an IF block and encloses a series of statements that are conditionally executed.
ELSE IF	F77	Begins an ELSE IF block of an IF block series and encloses statements that are conditionally executed.
ELSE WHERE	F90	The portion of the WHERE ELSE WHERE construct that permits conditional masked assignments to the elements of an array or to a scalar, zero-dimensional array.
ENCODE	F77 ext	Transfers data between variables or arrays in internal storage and translates that data from internal to character form, according to format specifiers.
END	F77	Terminates a segment of a Fortran program.
END ASSOCIATE	F2003	Terminates an Associate block.
END DO	F77	Terminates a DO or DO WHILE loop.
END FILE	F77	Writes an endfile record to the files.
END IF	F77	Terminates an IF ELSE or ELSE IF block.
END MAP	F77 ext	Terminates a MAP declaration.
END SELECT	F90	Terminates a SELECT declaration.
END STRUCTURE	F77 ext	Terminates a STRUCTURE declaration.
END UNION	F77 ext	Terminates a UNION declaration.
END WHERE	F90	Terminates a WHERE ELSE WHERE construct.
ENTRY	F77	Allows a subroutine or function to have more than one entry point.
EQUIVALENCE	F77	Allows two or more named regions of data memory to share the same start address.
EXIT	F90	Interrupts a DO construct execution and continues with the next statement after the loop.
EXTERNAL	F77	Identifies a symbolic name as an external or dummy procedure which can then be used as an actual argument.
FINAL	F2003	Specifies a Final subroutine inside a derived type.
FORALL	F95	Provides, as a statement or construct, a parallel mechanism to assign values to the elements of an array.
FORMAT	F77	Specifies format requirements for input or output.
FUNCTION	F77	Introduces a program unit; all the statements that follow apply to the function itself.
GENERIC	F2003	Specifies a generic type bound procedure inside a derived type.

Statement	Origin	Description
GOTO (Assigned)	F77	[Obsolescent]. Transfers control so that the statement identified by the statement label is executed next.
GOTO (Computed)	F77	Transfers control to one of a list of labels according to the value of an expression.
GOTO (Unconditional)	F77	Unconditionally transfers control to the statement with the label *label*, which must be declared within the code of the program unit containing the GOTO statement and must be unique within that program unit.
IF (Arithmetic)	F77	[Obsolescent]. Transfers control to one of three labeled statements, depending on the value of the arithmetic expression.
IF (Block)	F77	Consists of a series of statements that are conditionally executed.
IF (Logical)	F77	Executes or does not execute a statement based on the value of a logical expression.
IMPLICIT	F77	Redefines the implied data type of symbolic names from their initial letter, overriding implied data types.
IMPORT	F2003	Gives access to the named entities of the containing scope.
INCLUDE	F77 ext	Directs the compiler to start reading from another file.
INQUIRE	F77	Inquires about the current properties of a particular file or the current connections of a particular unit.
INTEGER	F77	Establishes the data type of a variable by explicitly attaching the name of a variable to an integer data type, overriding implied data types.
INTENT	F90	Specifies intended use of a dummy argument, but may not be used in a main program's specification statement.
INTERFACE	F90	Makes an implicit procedure an explicit procedure where the dummy parameters and procedure type are known to the calling module; Also overloads a procedure name.
INTRINSIC	F77	Identifies a symbolic name as an intrinsic function and allows it to be used as an actual argument.
LOGICAL	F77	Establishes the data type of a variable by explicitly attaching the name of a variable to a logical data type, overriding implied data types.
MAP	F77 ext	Designates each unique field or group of fields within a UNION statement.
MODULE	F90	Specifies the entry point for a Fortran 90/95 module program unit. A module defines a host environment of scope of the module, and may contain subprograms that are in the same scoping unit.
NAMELIST	F90	Allows the definition of namelist groups for namelist-directed I/O.
NULLIFY	F90	Disassociates a pointer from its target.

Statement	Origin	Description
OPEN	F77	Connects an existing file to a unit, creates and connects a file to a unit, creates a file that is preconnected, or changes certain specifiers of a connection between a file and a unit.
OPTIONAL	F90	Specifies dummy arguments that may be omitted or that are optional.
OPTIONS	F77 ext	Confirms or overrides certain compiler command-line options.
PARAMETER	F77	Gives a symbolic name to a constant.
PAUSE	F77	[Obsolescent]. Stops the program's execution.
POINTER	F90	Provides a means for declaring pointers.
POINTER (Cray)	F77 ext	Declares a scalar variable to be a pointer variable (of type INTEGER), and another variable to be its pointer-based variable.
PRINT	F77	Transfers data to the standard output device from the items specified in the output list and format specification.
PRIVATE	F90 F2003	Specifies entities defined in a module are not accessible outside of the module. `Private` can also appear inside a derived type to disallow access to its data components outside the defining module. In F2003, a `Private` statement may appear after the type's `contains` statement to disallow access to type bound procedures outside the defining module.
PROCEDURE	F2003	Specifies a type bound procedure, procedure pointer, module procedure, dummy procedure, intrinsic procedure, or an external procedure.
PROGRAM	F77	Specifies the entry point for the linked Fortran program.
PROTECTED	F2003	Protects a module variable against modification from outside the module in which it was declared.
PUBLIC	F90	Specifies entities defined in a module are accessible outside of the module.
PURE	F95	Indicates that a function or subroutine has no side effects.
READ	F90	Transfers data from the standard input device to the items specified in the input and format specifications.
REAL	F90	Establishes the data type of a variable by explicitly attaching the name of a variable to a data type, overriding implied data types.
RECORD	F77 ext	A VAX Fortran extension, defines a user-defined aggregate data item.
RECURSIVE	F90	Indicates whether a function or subroutine may call itself recursively.
REDIMENSION	F77 ext	Dynamically defines the bounds of a deferred-shape array.
RETURN	F77	Causes a return to the statement following a CALL when used in a subroutine, and returns to the relevant arithmetic expression when used in a function.

Statement	Origin	Description
REWIND	F77	Positions the file at its beginning. The statement has no effect if the file is already positioned at the start or if the file is connected but does not exist.
SAVE	F77	Retains the definition status of an entity after a RETURN or END statement in a subroutine or function has been executed.
SELECT CASE	F90	Begins a CASE construct.
SELECT TYPE	F2003	Provides the capability to execute alternative code depending on the dynamic type of a polymorphic entity and to gain access to dynamic parts. The alternative code is selected using the `type is` statement for a specific dynamic type, or the `class is` statement for a specific type and all its type extensions. Use the optional `class default` statement to specify all other dynamic types that don't match a specified `type is` or `class is` statement. Like the CASE construct, the code consists of a number of blocks and at most one is selected for execution.
SEQUENCE	F90	A derived type qualifier that specifies the ordering of the storage associated with the derived type. This statement specifies storage for use with COMMON and EQUIVALENCE statements.
STOP	F77	Stops the program's execution and precludes any further execution of the program.
STRUCTURE	F77 Vax ext	A VAX extension to FORTRAN 77 that defines an aggregate data type.
SUBROUTINE	F77	Introduces a subprogram unit.
TARGET	F90	Specifies that a data type may be the object of a pointer variable (e.g., pointed to by a pointer variable). Types that do not have the TARGET attribute cannot be the target of a pointer variable.
THEN	F77	Part of a block IF statement, surrounds a series of statements that are conditionally executed.
TYPE	F90 F2003	Begins a derived type data specification or declares variables of a specified user-defined type. Use the optional EXTENDS statement with TYPE to indicate a type extension in F2003.
UNION	F77 Vax ext	A multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields.
USE	F90	Gives a program unit access to the public entities or to the named entities in the specified module.
VOLATILE	F77 ext	Inhibits all optimizations on the variables, arrays and common blocks that it identifies.

Statement	Origin	Description
WAIT	F2003	Performs a wait operation for specified pending asynchronous data transfer operations.
WHERE	F90	Permits masked assignments to the elements of an array or to a scalar, zero dimensional array.
WRITE	F90	Transfers data to the standard output device from the items specified in the output list and format specification.

ACCEPT

The ACCEPT statement has the same syntax as the PRINT statement and causes formatted input to be read on standard input. ACCEPT is identical to the READ statement with a unit specifier of asterisk (*).

F77 extension

Syntax

```
ACCEPT f [,iolist]
ACCEPT namelist
```

f
format-specifier, a * indicates list directed input.

iolist
is a list of variables to be input.

namelist
is the name of a namelist specified with the NAMELIST statement.

Examples

```
 ACCEPT *, IA, ZA
 ACCEPT 99, I, J, K
 ACCEPT SUM
99 FORMAT(I2, I4, I3)
```

Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. In the following example, sum is treated as a format descriptor. The code in the first column is roughly equivalent to that in the second column.

```
real sum
sum = 4h()
accept sum
```

```
character*4 ch
ch = '()'
accept ch
```

See Also

READ, PRINT

ARRAY

The ARRAY attribute defines the number of dimensions in an array that may be defined and the number of elements and bounds in each dimension. [Not in PVF]

CMF

Syntax

```
ARRAY [::] array-name (array-spec)
 [, array-name (array-spec) ] ...
```

array-name
: is the symbolic name of an array.

array-spec
: is a valid array specification, either explicit-shape, assumed-shape, deferred-shape, or assumed size (refer to Chapter 4, "*Fortran Arrays*", for details on array specifications).

Description

ARRAY can be used in a subroutine as a synonym for DIMENSION to establish an argument as an array, and in this case the declarator can use expressions formed from integer variables and constants to establish the dimensions (adjustable arrays).

Note

These integer variables must be either arguments or declared in COMMON; they cannot be local. Further, in this case, the function of ARRAY statement is merely to supply a mapping of the argument to the subroutine code, and not to allocate storage.

The typing of the array in an ARRAY statement is defined by the initial letter of the array name in the same way as variable names, unless overridden by an IMPLICIT or type declaration statement. Arrays may appear in type declaration and COMMON statements but the array name can appear in only one array declaration.

Example

```
REAL, ARRAY(3:10):: ARRAY_ONE
INTEGER, ARRAY(3,-2:2):: ARRAY_TWO
```

This specifies ARRAY_ONE as a vector having eight elements with the lower bound of 3 and the upper bound of 10.

ARRAY_TWO as a matrix of two dimensions having fifteen elements. The first dimension has three elements and the second has five with bounds from -2 to 2.

See Also

ALLOCATE, DEALLOCATE

BYTE

The BYTE statement establishes the data type of a variable by explicitly attaching the name of a variable to a 1-byte integer. This overrides data typing implied by the initial letter of a symbolic name.

F77 extension

Syntax

```
BYTE name [/clist/],
...
```

name
: is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

clist
: is a list of constants that initialize the data, as in a DATA statement.

Description

Byte statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. BYTE declaration statements must not be labeled.

Example

```
BYTE TB3, SEC, STORE (5,5)
```

DECODE

The DECODE statement transfers data between variables or arrays in internal storage and translates that data from character form to internal form, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential READ statements.

F77 extension

Syntax

```
DECODE (c, f, b [ ,IOSTAT= ios ] [,
ERR= errs]) [ list ]
```

c
: is an integer expression specifying the number of bytes involved in translation.

f
: is the format-specifier.

b
: is a scalar or array reference for the buffer area containing formatted data (characters).

ios
: is an integer scalar memory reference which is the input/output status specifier: if this is specified ios becomes defined with zero if no error condition exists or a positive integer when there is an error condition.

errs
: an error specifier which takes the form of a statement label of an executable statement in the same program unit. If an error condition occurs execution continues with the statement specified by errs.

list
: is a list of input items.

Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. In the following example, `sum` is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

The preceding code segment is roughly equivalent to this:

```
character*4 ch
ch = '()'
accept ch
```

See Also

READ, PRINT,

DOUBLE COMPLEX

The DOUBLE COMPLEX statement establishes the data type of a variable by explicitly attaching the name of a variable to a double complex data type. This overrides the data typing implied by the initial letter of a symbolic name.

F77 extension

Syntax

The syntax for DOUBLE COMPLEX has two forms, a standard Fortran 90/95 entity based form, and the PGI extended form. This section describes both syntax forms.

```
DOUBLE COMPLEX [, attribute-list ::] entity-list
```

attribute-list
: is the list of attributes for the double complex variable.

entity-list
: is the list of defined entities.

Syntax Extension

```
DOUBLE COMPLEX name [/clist/] [,name] [/clist/]...
```

name
is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

clist
is a list of constants that initialize the data, as in a DATA statement.

Description

Type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

The default size of a DOUBLE COMPLEX variable is 16 bytes. With the -r8 option, the default size of a DOUBLE COMPLEX variable is also 16 bytes.

Examples

```
DOUBLE COMPLEX CURRENT, NEXT
```

See Also

COMPLEX

DOUBLE PRECISION

The DOUBLE PRECISION statement establishes the data type of a variable by explicitly attaching the name of a variable to a double precision data type. This overrides the data typing implied by the initial letter of a symbolic name.

F90

Syntax

The syntax for DOUBLE PRECISION has two forms, a standard Fortran 90/95 entity based form, and the PGI extended form. This section describes both syntax forms.

```
DOUBLE PRECISION [, attribute-list ::] entity-list
```

attribute-list
is the list of attributes for the double precision variable.

entity-list
is the list of defined entities.

Syntax Extension

```
DOUBLE PRECISION name [/clist/] [,
name] [/clist/]...
```

name
: is the symbolic name of a variable, array, or an array declarator (see the DIMENSION statement for an explanation of array declarators).

clist
: is a list of constants that initialize the data, as in a DATA statement.

Description

Type declaration statements may be used to dimension arrays explicitly in the same way as the DIMENSION statement. Type declaration statements must not be labeled. Note: The data type of a symbol may be explicitly declared only once. It is established by type declaration statement, IMPLICIT statement or by predefined typing rules. Explicit declaration of a type overrides any implicit declaration. An IMPLICIT statement overrides predefined typing rules.

The default size of a DOUBLE PRECISION variable is 8 bytes, with or without the `-r8` option.

Example

```
DOUBLE PRECISION PLONG
```

ENCODE

The ENCODE statement transfers data between variables or arrays in internal storage and translates that data from internal to character form, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential WRITE statements.

F77 extension

Syntax

```
ENCODE (c,f,b[,IOSTAT=ios] [,ERR=errs])[list]
```

c
: is an integer expression specifying the number of bytes involved in translation.

f
: is the format-specifier.

b
: is a scalar or array reference for the buffer area receiving formatted data (characters).

ios
: is an integer scalar memory reference which is the input/output status specifier: if this is included, ios becomes defined with zero if no error condition exists or a positive integer when there is an error condition.

errs
: an error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs execution continues with the statement specified by errs.

list
: a list of output items.

Non-character Format-specifier

If a format-specifier is a variable which is neither CHARACTER nor a simple INTEGER variable, the compiler accepts it and treats it as if the contents were character. For example, below sum is treated as a format descriptor:

```
real sum
sum = 4h()
accept sum
```

and is roughly equivalent to

```
character*4
ch
ch = '()'
accept ch
```

See Also

READ, PRINT

END MAP

The END MAP statement terminates a MAP declaration.

F77 extension

Syntax

```
END MAP
```

Description

For more information, refer to the "MAP" statement.

Example

```
MAP ! Structure for a customer
 INTEGER*4 acct_cust
 REAL*4 credit_amt
 CHARACTER*8 due_date
 END MAP
```

END STRUCTURE

The END STRUCTURE statement terminates a STRUCTURE declaration.

F77 extension

Syntax

```
END STRUCTURE
```

Description

For more information, refer to the "STRUCTURE" statement.

END UNION

The END UNION statement terminates a UNION declaration.

F77 extension

Syntax

```
END UNION
```

Description

For more information, refer to the "UNION" statement.

INCLUDE

The INCLUDE statement directs the compiler to start reading from another file.

Note

The INCLUDE statement is used for FORTRAN 77. There is no support for VAX/VMS text libraries or the module_name pathname qualifier that exists in the VAX/VMS version of the INCLUDE statement.

F77 extension

Syntax

```
INCLUDE 'filename [/[NO]LIST]'
INCLUDE "filename [/[NO]LIST]"
```

The following rules apply to the INCLUDE statement:

- The INCLUDE statement may be nested to a depth of 20 and can appear anywhere within a program unit as long as Fortran's statement-ordering restrictions are not violated.
- You can use the qualifiers /LIST and /NOLIST to control whether the include file is expanded in the listing file (if generated).

  ### Note

  There is no support for VAX/VMS text libraries or the module_name pathname qualifier that exists in the VAX/VMS version of the INCLUDE statement.

- Either single or double quotes may be used.
- If the final component of the file pathname is /LIST or /NOLIST, the compiler assumes it is a qualifier, unless an additional qualifier is supplied.
- The filename and the /LIST or /NOLIST qualifier may be separated by blanks.

The compiler searches for the include file in the following directories:

- Each –I directory specified on the command-line.

- The directory containing the file that contains the INCLUDE statement (the current working directory.)
- The standard include area.

Example

```
INCLUDE '/mypath/list /list'
```

This line includes a file named /mypath/list and expands it in the listing file, if a listing file is used.

MAP

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration. For more information on field alignment, refer to "Structures," on page 31.

F77 extension

Syntax

```
MAP
 field_declaration
 [field_declaration]
 ...
 [field_declaration]
END MAP
```

field_declaration
: is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union.

Description

Data can be initialized in field declaration statements in union declarations. However, it is illegal to initialize multiple map declarations in a single union.

The size of the shared area for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the language C requires one to associate a name with each map (union). Fortran field names must be unique within the same declaration nesting level of maps.

Example

The following is an example of RECORD, STRUCTURE and UNION usage. The size of each element of the recarr array would be the size of typetag (4 bytes) plus the size of the largest MAP (the employee map at 24 bytes).

```
STRUCTURE /account/
 INTEGER typetag ! Tag to determine defined map
 UNION
 MAP ! Structure for an employee
 CHARACTER*12 ssn ! Social Security Number
 REAL*4 salary
 CHARACTER*8 empdate ! Employment date
 END MAP
 MAP ! Structure for a customer
 INTEGER*4 acct_cust
 REAL*4 credit_amt
 CHARACTER*8 due_date
 END MAP
 MAP ! Structure for a supplier
 INTEGER*4 acct_supp
 REAL*4 debit_amt
 BYTE num_items
 BYTE items(12) ! Items supplied
 END MAP
 END UNION
END STRUCTURE
RECORD /account/ recarr(1000)
```

POINTER (Cray)

The POINTER statement is an extension to FORTRAN 77. It declares a scalar variable to be a pointer variable (of type INTEGER), and another variable to be its pointer-based variable.

F77 extension

Syntax

```
POINTER (p1, v1) [, (p2, v2) ...]
```

v1 and v2
: are pointer-based variables. A pointer-based variable can be of any type, including STRUCTURE. A pointer-based variable can be dimensioned in a separate type, in a DIMENSION statement, or in the POINTER statement. The dimension expression may be adjustable, where the rules for adjustable dummy arrays regarding any variables which appear in the dimension declarators apply.

p1 and p2
: are the pointer variables corresponding to v1 and v2. A pointer variable may not be an array. The pointer is an integer variable containing the address of a pointer-based variable. The storage located by the pointer variable is defined by the pointer-based variable (for example, array, data type, etc.). A reference to a pointer-based variable appears in Fortran statements like a normal variable reference (for example, a local variable, a COMMON block variable, or a dummy variable). When the based variable is referenced, the address to which it refers is always taken from its associated pointer (that is, its pointer variable is dereferenced).

The pointer-based variable does not have an address until its corresponding pointer is defined. The pointer is defined in one of the following ways:

- By assigning the value of the LOC function.
- By assigning a value defined in terms of another pointer variable.

- By dynamically allocating a memory area for the based variable. If a pointer-based variable is dynamically allocated, it may also be freed.

Example

```
REAL XC(10)
COMMON IC, XC
POINTER (P, I)
POINTER (Q, X(5))
P = LOC(IC)
I = 0        ! IC gets 0
P = LOC(XC)
Q = P + 20   ! same as LOC(XC(6))
X(1) = 0     ! XC(6) gets 0
ALLOCATE (X) ! Q locates a dynamically
             ! allocated memory area
```

Restrictions

The following restrictions apply to the POINTER statement:

- No storage is allocated when a pointer-based variable is declared.
- If a pointer-based variable is referenced, its pointer variable is assumed to be defined.
- A pointer-based variable may not appear in the argument list of a SUBROUTINE or FUNCTION and may not appear in COMMON, EQUIVALENCE, DATA, NAMELIST, or SAVE statements.
- A pointer-based variable can be adjusted only in a SUBROUTINE or FUNCTION subprogram. If a pointer-based variable is an adjustable array, it is assumed that the variables in the dimension declarator(s) are defined with an integer value at the time the SUBROUTINE or FUNCTION is called. For a variable which appears in a pointer-based variable's adjustable declarator, modifying its value during the execution of the SUBROUTINE or FUNCTION does not modify the bounds of the dimensions of the pointer-based array.
- A pointer-based variable is assumed not to overlap with another pointer-based variable.

PROTECTED

The PROTECTED statement protects a module variable against modification from outside the module in which it was declared.

F2003

Syntax

```
PROTECTED [ :: ], name [ , name ]
```

Description

Variables with the PROTECTED attribute may only be modified within the defining module. Outside of that module they are not allowed to appear in any variable definition context, that is, on the left-hand-side of an assignment statement.

This statement allows the values of variables of a module to be generally available without relinquishing control over their modification.

Examples

In the following module, the `cm_2_inch` and `in_2_cm` variables are protected so that they cannot be changed outside the CONVERT_FORMULA module. The PROTECTED attribute allows users of this module to read the measurements in either centimeters or inches, but the variables can only be changed via the provided subroutines which ensure that both values agree.

```
MODULE COVERT_FORMULA
REAL,PROTECTED :: in_2_cm = 2.54, cm_2_in = 0.39
CONTAINS
SUBROUTINE set_metric(new_value_cm)
...
END SUBROUTINE
SUBROUTINE set_english(new_value_in)
...
END SUBROUTINE
END MODULE
```

RECORD

The RECORD statement, a VAX Fortran extension, defines a user-defined aggregate data item.

F77 extension

Syntax

```
RECORD /structure_name/record_namelist
 [,/structure_name/record_namelist]
 ...
 [,/structure_name/record_namelist]
END RECORD
```

structure_name
: is the name of a previously declared structure.

record_namelist
: is a list of one or more variable or array names separated by commas.

Description

You create memory storage for a record by specifying a structure name in the RECORD statement. You define the field values in a record either by defining them in the structure declaration or by assigning them with executable code.

You can access individual fields in a record by combining the parent record name, a period (.), and the field name (for example, *recordname.fieldname*). For records, a scalar reference means a reference to a name that resolves to a single typed data item (for example, INTEGER), while an aggregate reference means a reference that resolves to a structured data item.

Scalar field references may appear wherever normal variable or array elements may appear with the exception of the COMMON, SAVE, NAMELIST, DATA and EQUIVALENCE statements. Aggregate references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms.

Records are allowed in COMMON and DIMENSION statements.

Example

```
STRUCTURE /PERSON/ ! Declare a structure
defining a person
 INTEGER ID
 LOGICAL LIVING
 CHARACTER*5 FIRST, LAST, MIDDLE
 INTEGER AGE
END STRUCTURE
 ! Define population to be an array where each element is of
 ! type person. Also define a variable, me, of type person.
RECORD /PERSON/ POPULATION(2), ME
 ...
ME.AGE = 34        ! Assign values for the variable me
ME.LIVING = .TRUE. ! to some of the fields.
ME.FIRST = 'Steve'
ME.ID = 542124822
 ...
POPULATION(1).LAST = 'Jones' ! Assign the "LAST" field of
                             ! element 1 of array population.
POPULATION(2) = ME   ! Assign all the values of record
                     ! "ME" to the record population(2)
```

REDIMENSION

The REDIMENSION statement, a PGF77 extension to FORTRAN 77, dynamically defines the bounds of a deferred-shape array. After a REDIMENSION statement, the bounds of the array become those supplied in the statement, until another such statement is encountered.

F77 extension

Syntax

```
REDIMENSION name ([lb:]ub[,[lb:]ub]...)
 [,name([lb:]ub[,[lb:]ub]...)]...
```

Where:

name
: is the symbolic name of an array.

[lb:]ub
: is a dimension declarator specifying the bounds for a dimension (the lower bound lb and the upper bound ub). lb and ub must be integers with ub greater than lb. The lower bound lb is optional; if it is not specified, it is assumed to be 1. The number of dimension declarations must be the same as the number of dimensions in the array.

Example

```
REAL A(:, :)
POINTER (P, A)
P = malloc(12 * 10 * 4)
REDIMENSION A(12, 10)
A(3, 4) = 33.
```

RETURN

The RETURN statement causes a return to the statement following a CALL when used in a subroutine, and returns to the relevant arithmetic expression when used in a function.

F77

Syntax

```
RETURN
```

Alternate RETURN

(Obsolescent) The alternate RETURN statement is obsolescent for HPF and Fortran 90/95. Use the CASE statement where possible in new or updated code. The alternate RETURN statement takes the following form:

```
RETURN expression
```

expression
: expression is converted to integer if necessary (expression may be of type integer or real). If the value of expression is greater than or equal to 1 and less than or equal to the number of asterisks in the SUBROUTINE or subroutine ENTRY statement then the value of expression identifies the nth asterisk in the actual argument list and control is returned to that statement.

Example

```
 SUBROUTINE FIX (A,B,*,*,C)
40 IF (T) 50, 60, 70
50 RETURN
60 RETURN 1
70 RETURN 2
 END
 PROGRAM FIXIT
 CALL FIX(X, Y, *100, *200, S)
 WRITE(*,5) X, S ! Come here if (T) < 0
 STOP
100 WRITE(*, 10) X, Y ! Come here if (T) = 0
 STOP
200 WRITE(*,20) Y, S ! Come here if (T) > 0
```

STRUCTURE

The STRUCTURE statement, a VAX extension to FORTRAN 77, defines an aggregate data type.

F77 VAX extension

Syntax

```
STRUCTURE [/structure_name/][field_namelist]
 field_declaration
 [field_declaration]
 ...
```

```
 [field_declaration]
END STRUCTURE
```

structure_name
: is unique and is used both to identify the structure and to allow its use in subsequent RECORD statements.

field_namelist
: is a list of fields having the structure of the associated structure declaration. A field_namelist is allowed only in nested structure declarations.

field_declaration
: can consist of any combination of substructure declarations, typed data declarations, union declarations or unnamed field declarations.

Description

Fields within structures conform to machine-dependent alignment requirements. Alignment of fields also provides a C-like "struct" building capability and allows convenient inter-language communications. Note that aligning of structure fields is not supported by VAX/VMS Fortran.

Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict. Also, because records use periods to separate fields, it is not legal to use relational operators (for example, .EQ., .XOR.), logical constants (.TRUE. or .FALSE.), or logical expressions (.AND., .NOT., .OR.) as field names in structure declarations.

Fields in a structure are aligned as required by hardware and a structure's storage requirements are therefore machine-dependent. Note that VAX/VMS Fortran does no padding. Because explicit padding of records is not necessary, the compiler recognizes the %FILL intrinsic, but performs no action in response to it.

Data initialization can occur for the individual fields.

The UNION and MAP statements are supported.

The following is an example of record and structure usage.

```
STRUCTURE /account/
 INTEGER typetag  ! Tag to determine defined map
 UNION
 MAP              ! Structure for an employee
 CHARACTER*12 ssn    ! Social Security Number
 REAL*4 salary
 CHARACTER*8 empdate ! Employment date
 END MAP
 MAP              ! Structure for a customer
 INTEGER*4 acct_cust
 REAL*4 credit_amt
 CHARACTER*8 due_date
 END MAP
 MAP          ! Structure for a supplier
 INTEGER*4 acct_supp
 REAL*4 debit_amt
 BYTE num_items
 BYTE items(12)   ! Items supplied
 END MAP
 END UNION
END STRUCTURE
```

```
RECORD /account/ recarr(1000)
```

UNION

A UNION declaration, a DEC extension to FORTRAN 77, is a multi-statement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. It declares groups of fields that share a common location within a structure. Each group of fields within a union declaration is declared by a map declaration, with one or more fields per map declaration.

Union declarations are used when one wants to use the same area of memory to alternately contain two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in a program, that field and any other fields in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

A union declaration is initiated by a UNION statement and terminated by an END UNION statement. Enclosed within these statements are one or more map declarations, initiated and terminated by MAP and END MAP statements, respectively. Each unique field or group of fields is defined by a separate map declaration. The format of a UNION statement is as follows:

F77 extension

Syntax

```
UNION
 map_declaration
 [map_declaration]
 ...
 [map_declaration]
END UNION
```

The format of the map_declaration is as follows:

```
MAP
 field_declaration
 [field_declaration]
 ...
 [field_declaration]
END MAP
```

field_declaration
: where field declaration is a structure declaration or RECORD statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union.

Description

Data can be initialized in field declaration statements in union declarations. Note, however, it is illegal to initialize multiple map declarations in a single union.

The size of the shared area for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it plus the space reserved for alignment purposes.

Manipulating data using union declarations is similar to using EQUIVALENCE statements. However, union declarations are probably more similar to union declarations for the language C. The main difference is that the language C requires one to associate a name with each map (union). Fortran field names must be unique within the same declaration nesting level of maps.

The following is an example of RECORD, STRUCTURE and UNION usage. The size of each element of the recarr array would be the size of typetag (4 bytes) plus the size of the largest MAP (the employee map at 24 bytes).

```
STRUCTURE /account/
 INTEGER typetag     ! Tag to determine defined map.
 UNION
 MAP                 ! Structure for an employee
 CHARACTER*12 ssn       ! Social Security Number
 REAL*4 salary
 CHARACTER*8 empdate    ! Employment date
 END MAP
 MAP                ! Structure for a customer
 INTEGER*4 acct_cust
 REAL*4 credit_amt
 CHARACTER*8 due_date
 END MAP
 MAP                ! Structure for a supplier
 INTEGER*4 acct_supp
 REAL*4 debit_amt
 BYTE num_items
 BYTE items(12)         ! Items supplied
 END MAP
 END UNION
END STRUCTURE
RECORD /account/ recarr(1000)
```

VOLATILE

The VOLATILE statement inhibits all optimizations on the variables, arrays and common blocks that it identifies. The VOLATILE attribute, added in Fortran 2003, is used in a type declaration statement.

F77 extension (statement)

F2003 (attribute)

Syntax

Volatile Attribute

```
   datatype, volatile :: var_name
OR
   datatype :: var_name
   volatile :: var_name
   var_name
```

Volatile Statement

```
VOLATILE nitem [, nitem ...]
```

nitem
: is the name of a variable, an array, or a common block enclosed in slashes.

Description

Being volatile indicates to the compiler that, at any time, the variable might change or be examined from outside the Fortran program. The impact on the programmer is that anytime a volatile variable is referenced, the value must be loaded from memory. Furthermore, any assignment to the volatile variable must be written to memory.

If *nitem* names a common block, all members of the block are volatile. The volatile attribute of a variable is inherited by any direct or indirect equivalences, as shown in the example.

Volatile Attribute Example

The following example declares both the integer variable `xyz` and the real variable `abc` to be volatile.

```
   integer, volatile :: xyz
   real :: abc
   volatile :: abc
```

Volatile Statement Example

```
COMMON /COM/ C1, C2
VOLATILE /COM/, DIR    ! /COM/ and DIR are volatile
EQUIVALENCE (DIR, X)   ! X is volatile
EQUIVALENCE (X, Y )    ! Y is volatile
```

WAIT

Performs a wait operation for specified pending asynchronous data transfer operations.

F2003

Syntax

```
WAIT (wait_specs_list)
```

`wait_specs_list` can include any of the following specifiers:

UNIT =] file-unit-number
: A file-unit-number must be specified. If the optional characters UNIT= are omitted, the file-unit-number is the first item in the wait-spec-list.

END = label
: `label` must be the statement label of a branch target statement that appears in the same scoping unit as the WAIT statement.

  END= specifier has no effect if the pending data transfer operation is not a READ.

EOR = label
: `label` must be the statement label of a branch target statement that appears in the same scoping unit as the WAIT statement.

  EOR= specifier has no effect if the pending data transfer operation is not a nonadvancing READ.

ERR = label
: `label` must be the statement label of a branch target statement that appears in the same scoping unit as the WAIT statement.

ID = scalar_int_var
: `scalar_int_var` is the identifier of a pending data transfer operation for the specified unit.

- If the ID= specifier appears, a wait operation for the specified data transfer operation is performed.
- If the ID= specifier is omitted, wait operations for all pending data transfers for the specified unit are performed.

IOMSG = iomsg-var
: `iomsg-var` is an I/O message variable.

IOSTAT =scalar-int-var
: **scalar_int_var** is the identifier of a pending data transfer operation for the specified unit.

For more information on IOSTAT, ERR=, EOR=, END=, and IOMSG=, refer to the READ and WRITE statements.

Description

This statement performs a wait operation for specified pending asynchronous data transfer operations.

The CLOSE, INQUIRE, and file positioning statements may also perform wait operations.

Execution of a WAIT statement specifying a unit that does not exist, has no file connected to it, or that was not opened for asynchronous input/output is permitted, provided that the WAIT statement has no ID= specifier. This type of WAIT statement does not cause an error or end-of-file condition to occur.

Note

No specifier shall appear more than once in a given wait-spec-list.

Examples

```
INTEGER SCORE(30)
CHARACTER GRADE(30)
WHERE ( SCORE > 60 )
 GRADE = 'P'
ELSE WHERE
 GRADE = 'F'
END WHERE
```

Chapter 4. Fortran Arrays

Fortran arrays are any object with the dimension attribute. In Fortran 90/95, arrays may be very different from arrays in older versions of Fortran. Arrays can have values assigned as a whole without specifying operations on individual array elements, and array sections can be accessed. Also, allocatable arrays that are created dynamically are available as part of the Fortran 90/95 standard. This chapter describes some of the features of Fortran 90/95 arrays.

The following example illustrates valid array operations.

```
REAL(10,10) A,B,C
A=12 !Assign 12 to all elements of A
B=3 !Assign 3 to all elements of B
C=A+B !Add each element of A to each of B
```

Array Types

Fortran supports four types of arrays: explicit-shape arrays, assumed-shape arrays, deferred-shape arrays and assumed-size arrays. Both explicit-shape arrays and deferred shape arrays are valid in a main program. Assumed shape arrays and assumed size arrays are only valid for arrays used as dummy arguments. Deferred shape arrays, where the storage for the array is allocated during execution, must be declared with either the ALLOCATABLE or POINTER attributes.

Every array has properties of type rank, shape and size. The extent of an array's dimension is the number of elements in the dimension. The array rank is the number of dimensions in the array, up to a maximum of seven. The shape is the vector representing the extents for all dimensions. The size is the product of the extents. For some types of arrays, all of these properties are determined when the array is declared. For other types of arrays, some of these properties are determined when the array is allocated or when a procedure using the array is entered. For arrays that are dummy arguments, there are several special cases.

Allocatable arrays are arrays that are declared but for which no storage is allocated until an allocate statement is executed when the program is running. Allocatable arrays provide Fortran 90/95 programs with dynamic storage. Allocatable arrays are declared with a rank specified with the ":" character rather than with explicit extents, and they are given the ALLOCATABLE attribute.

Explicit Shape Arrays

Explicit shape arrays are those arrays familiar to FORTRAN 77 programmers. Each dimension is declared with an explicit value. There are two special cases of explicit arrays. In a procedure, an explicit array whose bounds

are passed in from the calling program is called an automatic-array. The second special case, also found in a procedure, is that of an adjustable-array which is a dummy array where the bounds are passed from the calling program.

Assumed Shape Arrays

An assumed shape array is a dummy array whose bounds are determined from the actual array. Intrinsics called from the called program can determine sizes of the extents in the called program's dummy array.

Deferred Shape Arrays

A deferred shape array is an array that is declared, but not with an explicit shape. Upon declaration, the array's type, its kind, and its rank (number of dimensions) are determined. Deferred shape arrays are of two varieties, allocatable arrays and array pointers.

Assumed Size Arrays

An assumed size array is a dummy array whose size is determined from the corresponding array in the calling program. The array's rank and extents may not be declared the same as the original array, but its total size (number of elements) is the same as the actual array. This form of array should not need to be used in new Fortran programs.

Array Specification

Arrays may be specified in either of two types of data type specification statements, attribute-oriented specifications or entity-oriented specifications. Arrays may also optionally have data assigned to them when they are declared. This section covers the basic form of entity-based declarations for the various types of arrays. Note that all the details of array passing for procedures are not covered here; refer to The Fortran 95 Handbook for complete details on the use of arrays as dummy arguments.

Explicit Shape Arrays

Explicit shape arrays are defined with a specified rank, each dimension must have an upper bound specified, and a lower bound may be specified. Each bound is explicitly defined with a specification of the form:

```
[lower-bound:] upper-bound
```

An array has a maximum of seven dimensions. The following are valid explicit array declarations:

```
INTEGER NUM1(1,2,3)           ! Three dimensions
INTEGER NUM2(-12:6,100:1000)  ! Two dimensions with lower & upper bounds
INTEGER NUM3(0,12,12,12)      ! Array of size 0
INTEGER NUM3(M:N,P:Q,L,99)    ! Array with 4 dimensions
```

Assumed Shape Arrays

An assumed shape array is always a dummy argument. An assumed shape array has a specification of the form:

```
[lower-bound] :
```

The number of colons (:) determines the array's rank. An assumed shape array cannot be an ALLOCATABLE or POINTER array.

Deferred Shape Arrays

An deferred shape array is an array pointer or an allocatable array. An assumed shape array has a specification that determines the array's rank and has the following form for each dimension:

```
:
```

For example:

```
INTEGER, POINTER::NUM1(:,:,:,:)
INTEGER, ALLOCATABLE::NUM2(:)
```

Assumed Size Arrays

An assumed size array is a dummy argument with an assumed size. The array’s rank and bounds are specified with a declaration that has the following form:

```
[explicit-shape-spec-list ,][lower-bound:]*
```

For example:

```
SUBROUTINE YSUM1(M,B,C)
 INTEGER M
 REAL, DIMENSION(M,4,5,*) :: B,C
```

Array Subscripts and Access

There are a variety of ways to access an array in whole or in part. Arrays can be accessed, used, and assigned to as whole arrays, as elements, or as sections. Array elements are the basic access method.

In the following example, the value of 5 is assigned to element 3,1 of NUMB.

```
INTEGER, DIMENSION(3,11) :: NUMB
NUMB(3,1)=5
```

The following statement assigns the value 5 to all elements of NUMB.

The array NUMB may also be accessed as an entire array:

```
NUMB=5
```

Array Sections and Subscript Triplets

Another possibility for accessing array elements is the array section. An array section is an array accessed by a subscript that represents a subset of the entire array's elements and is not an array element. An array section resulting from applying a subscript list may have a different rank than the original array. An array section's subscript list consists of subscripts, subscript triplets, and/or vector subscripts.

The following example uses a subscript triplet and a subscript, assigning the value 6 to all elements of NUMB with the second dimension of value 3 (NUMB(1,3), NUMB(2,3), NUMB(3,3)).

```
NUMB(:,3)=6
```

The following array section uses the array subscript triplet and a subscript to access three elements of the original array. This array section could also be assigned to a rank one array with three elements, as shown here:

```
INTEGER(3,11) NUMB
INTEGER(3) NUMC
NUMB(:,3)=6
NUMC=NUMB(:,3)
```

In this example, NUMC is rank 1 and NUMB is rank 2. This assignment, using the subscript 3, illustrates how NUMC, and the array section of NUMB, has a shape that is of a different rank than the original array.

The general form for an array's dimension with a vector subscript triplet is:

```
[subscript] : [subscript] [:stride]
```

The first subscript is the lower bound for the array section, the second is the upper bound and the third is the stride. The stride is by default one. If all values except the : are omitted, then all the values for the specified dimensions are included in the array section.

In the following example, using the NUMB previously defined, the statement has a stride of 2, and assigns the value 7 to the elements NUMB(1,3) and NUMB(3,3).

```
NUMB(1:3:2,3)=7
```

Array Sections and Vector Subscripts

Vector-valued subscripts specify an array section by supplying a set of values defined in a one dimensional array (vector) for a dimension or several dimensions of an array section.

In the following example, the array section uses the vectors I and J to assign the value 7 to each of the elements: NUMB(2,1), NUMB(2,2), NUMB(3,1), and NUMB(3,2).

```
INTEGER J(2), I(2)
INTEGER NUMB(3,6)
I=(/1,2/)
J=(/2,3/)
NUMB(J,I)=7
```

Array Constructors

An array constructor can be used to assign values to an array. Array constructors form one-dimensional vectors to supply values to a one-dimensional array, or one dimensional vectors and the RESHAPE function to supply values to arrays with more than one dimension.

Array constructors can use a form of implied DO similar to that in a DATA statement. For example:

```
INTEGER DIMENSION(4):: K = (/1,2,7,11/)
INTEGER DIMENSION(20):: J = (/(I,I=1,40,2)/)
```

Chapter 5. Input and Output

Input, output, and format statements provide the means for transferring data to or from files. Data is transferred as records to or from files. A record is a sequence of data which may be values or characters and a file is a sequence of such records. A file may be internal, that is, held in memory, or external such as those held on disk. To access an external file a formal connection must be made between a unit, for example a disk file, and the required file. An external unit must be identified either by a positive integer expression, the value of which indicates a unit, or by an asterisk (*) which identifies a standard input or output device.

This chapter describes the types of input and output available and provides examples of input, output and format statements. There are four types of input/output used to transfer data to or from files: unformatted, formatted, list directed, and namelist.

- unformatted data is transferred between the item(s) in the input/output list (iolist) and the current record in the file. Exactly one record may be read or written.
- formatted data is edited to conform to a format specification, and the edited data is transferred between the item or items in the iolist, and the file. One or more records may be read or written. Non-advancing formatted data transfers are a variety of formatted I/O where a portion of a data record is transferred with each input/output statement.
- list directed input/output is an abbreviated form of formatted input/output that does not use a format specification. Depending on the type of the data item or data items in the iolist, data is transferred to or from the file, using a default, and not necessarily accurate format specification.
- namelist input/output is a special type of formatted data transfer; data is transferred between a named group (namelist group) of data items and one or more records in a file.

File Access Methods

You can access files using one of two methods, sequential access, or direct access (random access). The access method is determined by the specifiers supplied when the file is opened using the OPEN statement. Sequential access files are accessed one after the other, and are written in the same manner. Direct access files are accessed by specifying a record number for input, and by writing to the currently specified record on output.

Files may contain one of two types of records, fixed length records or variable length records. To specify the size of the fixed length records in a file, use the RECL specifier with the OPEN statement. RECL sets the record length in bytes.[1] RECL can only be used when access is direct.

A record in a variable length formatted file is terminated with \n. A record in a variable length unformatted file is preceded and followed by a word indicating the length of the record.

Standard Preconnected Units

Certain input and output units are predefined, depending on the value of compiler options. The PGI Fortran compilers `-Mdefaultunit` option tells the compiler to treat "*" as a synonym for standard input for reading and standard output for writing. When the option is `-Mnodefaultunit`, the compiler treats "*" as a synonym for unit 5 on input and unit 6 on output.

Opening and Closing Files

The OPEN statement establishes a connection to a file. OPEN allows you to do any of the following

- Connect an existing file to a unit.
- Create and connect a file to a unit.
- Create a file that is preconnected.
- Establish the access method and record format for a connection.

OPEN has the form:

```
OPEN (list)
```

where list contains a unit specifier of the form:

```
[UNIT=] u
```

where u, an integer, is the external unit specifier.

In addition list may contain one of each of the specifiers shown in Table 5.1, “OPEN Specifiers”.

Direct Access Files

If a file is connected for direct access using OPEN with ACCESS='DIRECT', the record length must be specified using RECL=. Further, one of each of the other specifiers may also be used.

Any file opened for direct access must be via fixed length records.

In the following example:

- A new file, `book.dat`, is created and connected to unit 12 for direct formatted input/output with a record length of 98 characters.

[1]The units depend on the value of the FORTRANOPT environment variable. If the value is vaxio, then the record length is in units of 32-bit words. If FORTRANOPT is not defined, or its value is something other than vaxio, then the record length is always in units of bytes.

- Blank values are ignored in numeric values.
- If an error condition exists when the OPEN statement is executed, the variable E1 is assigned some positive value, and then execution continues with the statement labeled 20.
- If no error condition pertains, E1 is assigned the value 0 and execution continues with the statement following the OPEN statement.

```
OPEN(12,IOSTAT=E1,ERR=20,FILE='book.dat',BLANK='NULL',
+ACCESS='DIRECT',RECL=98,FORM='FORMATTED',STATUS='NEW')
```

Closing a File

Close a unit by specifying the CLOSE statement from within any program unit. If the unit specified does not exist or has no file connected to it, the CLOSE statement has no effect.

Provided the file is still in existence, it may be reconnected to the same or a different unit after the execution of a CLOSE statement. An implicit CLOSE is executed when a program stops.

The CLOSE statement terminates the connection of the specified file to a unit.

```
CLOSE ([UNIT=] u [,IOSTAT=ios] [,ERR= errs ]
[,STATUS= sta] [,DISPOSE= sta] [,DISP= sta])
```

CLOSE takes the status values IOSTAT, ERR, and STATUS, similar to those described in Table 5.1, "OPEN Specifiers". In addition, CLOSE allows the DISPOSE or DISP specifier which can take a status value sta which is a character string, where case is insignificant, specifying the file status (the same keywords are used for the DISP and DISPOSE status). Status can be KEEP or DELETE. KEEP cannot be specified for a file whose dispose status is SCRATCH. When KEEP is specified (for a file that exists) the file continues to exist after the CLOSE statement, conversely DELETE deletes the file after the CLOSE statement. The default value is KEEP unless the file status is SCRATCH.

Table 5.1. OPEN Specifiers

Specifier	Description
ACCESS=acc	Where acc is a character string specifying the access method for file connection as DIRECT (random access) or SEQUENTIAL. The default is SEQUENTIAL.
ACTION=act	Where act is a character string specifying the allowed actions for the file and is one of READ, WRITE, or READWRITE.
ASYNCHRONOUS=async	Where async is a character expression specifying whether to allow asynchronous data transfer on this file connection. One of 'YES' or 'NO' is allowed.
BLANK=blnk	Where blnk is a character string which takes the value NULL or ZERO: NULL causes all blank characters in numeric formatted input fields to be ignored with the exception of an all-blank field which has a value of zero. ZERO causes all blanks other than leading blanks to be treated as zeros. The default is NULL. This specifier must only be used when a file is connected for formatted input/output.

Specifier	Description
CONVERT=char_expr	Where char_expr is a character string that allows byte-swapping I/O to be performed on specific logical units, and is one of following: BIG_ENDIAN, LITTLE_ENDIAN, or NATIVE. Previously, byte-swapping I/O was only enabled by the command-line option, `-byteswapio`, and was applied to all unformatted I/O operations which appeared in the files compiled using `-byteswapio`. The value 'BIG_ENDIAN' is specifies to convert big-endian format data files produced by most RISC workstations and high-end servers to the little-endian format used on Intel Architecture systems on-the-fly during file reads/writes. This value assumes that the record layouts of unformatted sequential access and direct access files are the same on the systems. For the values 'LITTLE_ENDIAN' and 'NATIVE", byte-swapping is not performed during file reads/writes since the little-endian format is used on Intel Architecture.
DECIMAL= scalar_char	Specify the default decimal edit mode for the unit. When the edit mode is *point*, decimal points appear in both input and output. The options are *COMMA*, where commas rather than decimal points appear in both input and output, and *POINT*, where decimal points appear in both input and output.
DELIM=del	Specify the delimiter for character constants written by a list-directed or namelist-formatted statement. The options are APOSTROPHE, QUOTE, and NONE.
ENCODING= specifier	An encoding specifier which indicates the desired encoding of the file, such as one of the following: `UTF-8` specifies the file is connected for UTF-8 I/O or that the processor can detect this format in some way. A processor-dependent value indicates the file is in another known format, such as `UTF-16LE`.
ERR=errs	An error specifier which takes the form of a statement label of an executable statement in the same program. If an error condition occurs, execution continues with the statement specified by errs.*2*
FILE=fin	Where fin is a character string defining the file name to be connected to the specified unit.
FORM=fm	Where fm is a character string specifying whether the file is being connected for FORMATTED, UNFORMATTED, or BINARY output. The default is FORMATTED. For an unformatted file whose form is BINARY, the file is viewed as a byte-stream file, such as a file created by fwrite() calls in a C program; the data in the file is not organized into records.

Specifier	Description
IOSTAT=ios	Input/output status specifier where ios is an integer scalar memory reference. If this is included in list, ios becomes defined with 0 if no error exists or a positive integer when there is an error condition.[a]
PAD=padding	Specifies whether or not to use blank padding for input items. The padding values are YES and NO. The value NO requires that the input record and the input list format specification match.
POSITION=pos	Specifies the position of an opened file. ASIS indicates the file position remains unchanged. REWIND indicates the file is to be rewound, and APPEND indicates the file is to positioned just before an end-of-file record, or at its terminal point.
RECL=rl	Where rl is an integer which defines the record length in a file connected for direct access and is the number of characters when formatted input/output is specified. This specifier must only be given when a file is connected for direct access.
Round=specifier	Where specifier is a character expression that controls the optional plus characters in formatted numeric output. The value can be SUPPRESS, PLUS, PROCESSOR_DEFINED, or UNDEFINED.
STATUS=sta	The file status where sta is a character expression: it can be NEW, OLD, SCRATCH, REPLACE or UNKNOWN. When OLD or NEW is specified a file specifier must be given. SCRATCH must not be used with a named file. The default is UNKNOWN.
SIGN=specifier	Where specifier is a character expression that controls the optional plus characters in formatted numeric output. The value can be SUPPRESS, PLUS, PROCESSOR_DEFINED, or UNDEFINED.

[a]If IOSTAT and ERR are not present, the program terminates if an error occurs.

A unit may be the subject of a CLOSE statement from within any module. If the unit specified does not exist or has no file connected to it, the use of the CLOSE statement has no effect. Provided the file is still in existence it may be reconnected to the same or a different unit after the execution of a CLOSE statement. Note that an implicit CLOSE is executed when a program stops.

In the following example the file on UNIT 6 is closed and deleted.

```
CLOSE(UNIT=6,STATUS='DELETE')
```

Data Transfer Statements

Once a unit is connected, either using a preconnection, or by executing an OPEN statement, data transfer statements may be used. The available data transfer statements include: READ, WRITE, and PRINT. The general form for these data transfer statements is shown in Chapter 3, "*Fortran Statements*"; refer to that section for details on the READ, WRITE and PRINT statements and their valid I/O control specifiers.

Unformatted Data Transfer

Unformatted data transfer allows data to be transferred between the current record and the items specified in an input/output list. Use OPEN to open a file for unformatted output:

```
OPEN (2, FILE='new.dat', FORM='UNFORMATTED')
```

The unit specified must be an external unit.

After data is transferred, the file is positioned after the last record read or written, if there is no error condition or end-of-file condition set.

Note

Unformatted data transfer cannot be carried out if the file is connected for formatted input/output.

The following example shows an unformatted input statement:

```
READ (2, ERR=50) A, B
```

- On output to a file connected for direct access, the output list must not specify more values than can fit into a record. If the values specified do not fill the record the rest of the record is undefined.
- On input, the file must be positioned so that the record read is either an unformatted record or an endfile record.
- The number of values required by the input list in the input statement must be less than or equal to the number of values in the record being read. The type of each value in the record must agree with that of the corresponding entity in the input list. However one complex value may correspond to two real list entities or vice versa. If the input list item is of type CHARACTER, its length must be the same as that of the character value
- In the event of an error condition, the position of the file is indeterminate.

Formatted Data Transfer

During formatted data transfer, data is edited to conform to a format specification, and the edited data is transferred between the items specified in the input or output statement's iolist and the file; the current record is read or written and, possibly, so are additional records. On input, the file must be positioned so that the record read is either a formatted record or an endfile record. Formatted data transfer is prohibited if the file is connected for unformatted input/output.

For variable length record formatted input, each newline character is interpreted as a record separator. On output, the I/O system writes a newline at the end of each record. If a program writes a newline itself, the single record containing the newline will appear as two records when read or backspaced over. The maximum allowed length of a record in a variable length record formatted file is 2000 characters.

Implied DO List Input Output List

An implied DO list takes the form

```
(iolist,do-var=var1,var2,var3)
```

where the items in iolist are either items permissible in an input/output list or another implied DO list. The value do-var is an INTEGER, REAL or DOUBLE PRECISION variable and var1, var2 and var3 are arithmetic expressions of type INTEGER, REAL or DOUBLE PRECISION. Generally, do-var, var1, var2 and var3 are of type INTEGER. Should iolist occur in an input statement, the do-var cannot be used as an item in iolist. If var3 and the preceding comma are omitted, the increment takes the value 1. The list items are specified once for each iteration of the DO loop with the DO-variable being substituted as appropriate.

In the following example OXO, C(7), C(8) and C(9) are each set to 0.0. TEMP, D(1) and D(2) are set to 10.0.

```
REAL C(6),D(6)
DATA OXO,(C(I),I=7,9),TEMP,(D(J),J=1,2)/4*0.0,3*10.0/
```

The following two statements have the same effect.

```
READ *,A,B,(R(I),I=1,4),S
```

```
READ *,A,B,R(1),R(2),R(3),R(4),S
```

Format Specifications

Format requirements may be given either in an explicit FORMAT statement or alternatively, as fields within an input/output statement (as values in character variables, arrays or other character expressions within the input/output statement).

When a format identifier in a formatted input/output statement is a character array name or other character expression, the leftmost characters must be defined with character data that constitute a format specification when the statement is executed. A character format specification is enclosed in parentheses. Blanks may precede the left parenthesis. Character data may follow the right-hand parenthesis and has no effect on the format specification. When a character array name is used as a format identifier, the length of the format specification can exceed the length of the first element of the array; a character array format specification is considered to be an ordered concatenation of all the array elements. When a character array element is used as a format identifier the length must not exceed that of the element used.

The FORMAT statement has the form:

```
FORMAT (list-of-format-requirements)
```

The list of format requirements can be any of the following, separated by commas:

- Repeatable editor commands which may or may not be preceded by an integer constant which defines the number of repeats.
- Non-repeatable editor commands.
- A format specification list enclosed in parentheses, optionally preceded by an integer constant which defines the number of repeats.

Each action of format control depends on a FORMAT specified edit code and the next item in the input/output list used. If an input/output list contains at least one item, there must be at least one repeatable edit code in the format specification. An empty format specification FORMAT( ) can only be used if no list items are specified. In such a case, one input record is skipped or an output record containing no characters is written. Unless the edit code or the format list is preceded by a repeat specification, a format specification is interpreted from left to right. When a repeat specification is used, the appropriate item is repeated the required number of times.

Each repeatable edit code has a corresponding item in the iolist; however when a list item is of type complex two edit codes of F, E, D or G are required. The edit codes P, X, T, TL, TR, S, SP, SS, H, BN, BZ, /, : and apostrophe act directly on the record and have no corresponding item in the input/output list.

The file is positioned after the last character read or written when the edit codes I, F, E, D, G, L, A, H or apostrophe are processed. If the specified unit is a printer then the first character of the record is used to control the vertical spacing as shown in the following table:

Table 5.2. Format Character Controls for a Printer

Character	Vertical Spacing
Blank	One line
0	Two lines
1	To first line on next page
+	No advance

A Format Control – Character Data

The A specifier transfers characters. The A specifier has the form:

```
Aw
```

When the optional width field, w, is not specified, the width is determined by the size of the data item.

On output, if l is the length of the character item and w is the field width, then the following rules apply:

- If w > l, output with w-l blanks before the character.
- If w < l, output leftmost w characters.

On input, if l is the length of the character I/O item and w is the field width, then the following rules apply:

- If w > l, rightmost l characters from the input filed.
- If w < l, leftmost w characters from the input filed and followed by l – w blanks.

You can also use the A format specifier to process data types other than CHARACTER. For types other than CHARACTER, the number of characters supplied for input/output equals the size in bytes of the data allocated to the data type. For example, an INTEGER*4 value is represented with 4 characters and a LOGICAL*2 is represented with 2 characters.

The following shows a simple example that reads two CHARACTER arrays from the file `data.src`:

```
 CHARACTER STR1*8, STR2*12
 OPEN(2, FILE='data.src')
 READ(2, 10) STR1, STR2
10 FORMAT ( A8, A12 )
```

B Format Control – Binary Data

The B field descriptor transfers binary values and can be used with any integer data type. The edit descriptor has the form:

```
Bw[.m]
```

where w specifies the field width and m indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) binary characters only (0 or 1). An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the B field descriptor transfers the binary values of the corresponding I/O list element, right-justified, to an external field that is w characters long.

- If the value to be transmitted does not fill the field, leading spaces are inserted.
- If the value is too large for the field, the entire field is filled with asterisks.
- If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary.
- If m is zero, and the internal representation is zero, the external field is blank-filled.

D Format Control – Real Double Precision Data with Exponent

The D specifier transfers real values for double precision data with a representation for an exponent. The form of the D specifier is:

```
Dw.d
```

where w is the field width and d the number of digits in the fractional part.

For input, the same conditions apply as for the F specifier described later in this chapter.

For output, the scale factor k controls the decimal normalization. The scale factor k is the current scale factor specified by the most recent P format control.

- If one hasn't been specified, the default is zero (0).
- If *-d < k <= 0*, the output file contains leading zeros and *d*-|*k*| significant digits after the decimal point.
- If *0 < k < d+2,* there are 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 allowed.

For example:

```
 DOUBLE PRECISION VAL1
 VAL1 = 141.8835
 WRITE( *, 20) VAL1
20 FORMAT ( D10.4 )
```

produces the following:

```
0.1418D+03
```

d Format Control – Decimal specifier

The dc and dp descriptors, representing decimal comma and decimal point edit modes, respectively, are valid in format processing, such as in a FORMAT statement.

The specific edit mode takes effect immediately when encountered in formatting, and stays in effect until either another descriptor is encountered or until the end of the

E Format Control – Real Single Precision Data with Exponent

The E specifier transfers real values for single precision data with an exponent. The E format specifier has two basic forms:

```
Ew.d
Ew.dEe
```

where w is the field width, d the number of digits in the fractional part and e the number of digits to be printed in the exponent part.

For input the same conditions apply as for F editing.

For output the scale factor controls the decimal normalization as in the D specifier.

EN Format Control

The EN specifier transfers real values using engineering notation.

```
ENw.d
ENw.dEe
```

where w is the field width, d the number of digits in the fractional part and e the number of digits to be printed in the exponent part.

On output, the number is in engineering notation where the exponent is divisible by 3 and the absolute value of the significand is 1000 > |significand | 1. This format is the same as the E format descriptor, except for restrictions on the size of the exponent and the significand.

ES Format Control

The ES specifier transfers real values in scientific notation. The ES format specifier has two basic forms:

```
ESw.d
ESw.dEe
```

where w is the field width, d the number of digits in the fractional part and e the number of digits to be printed in the exponent part.

For output, the scale factor controls the decimal normalization as in the D specifier.

On output, the number is presented in scientific notation, where the absolute value of the significand is 10> | significand | 1.

F Format Control - Real Single Precision Data

The F specifier transfers real values. The form of the F specifier is:

```
Fw.d
```

where w is the field width and d is the number of digits in the fractional part.

On input, if the field does not contain a decimal digit or an exponent, right-hand d digits, with leading zeros, are interpreted as being the fractional part.

On output, a leading zero is only produced to the left of the decimal point if the value is less than one.

G Format Control

The G format specifier provides generalized editing of real data. The G format has two basic forms:

```
Gw.d
Gw.dEe
```

The specifier transfers real values; it acts like the F format control on input and depending on the value's magnitude, like E or F on output. The magnitude of the data determines the output format. For details on the actual format used, based on the magnitude, refer to the ANSI FORTRAN Standard (Section 13.5.9.2.3 G Editing).

I Format Control – Integer Data

The I format specifier transfers integer values. The I format specifier has two basic forms:

```
Iw
Iw.m
```

where w is the field width and m is the minimum filed width on output, including leading zeros. If present, m must not exceed width w.

On input, the external field to be input must contain (unsigned) decimal characters only. An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the I format descriptor transfers the decimal values of the corresponding I/O list element, right-justified, to an external field that is w characters long.

- If the value to be transmitted does not fill the field, leading spaces are inserted.
- If the value is too large for the field, the entire field is filled with asterisks.
- If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary.
- If m is zero, and the internal representation is zero, the external field is blank-filled.

L Format Control – Logical Data

The L format control transfers logical data of field width w:

```
Lw
```

On input, the list item will become defined with a logical value; the field consists of optional blanks, followed by an optional decimal point followed by T or F. The values .TRUE. or .FALSE. may also appear in the input field

The output field consists of w-1 blanks followed by T or F as appropriate.

Quote Format Control

Quote editing prints a character constant. The format specifier writes the characters enclosed between the quotes and cannot be used on input. The field width is that of the characters contained within quotes (you can also use apostrophes to enclose the character constant).

To write an apostrophe (or quote), use two consecutive apostrophes (or quotes).

For example:

```
 WRITE ( *, 101)
101 FORMAT ( 'Print an apostrophe '' and end.')
```

Produces:

```
Print an apostrophe ' and end.
```

Similarly, you can use quotes, for example:

```
 WRITE ( *, 102)
102 FORMAT ( "Print a line with a "" and end.")
```

Produces:

```
Print a line with a " and end.
```

BN Format Control – Blank Control

The BN and BZ formats control blank spacing. BN causes all embedded blanks except leading blanks in numeric input to be ignored, which has the effect of right-justifying the remainder of the field. Note that a field of all blanks has the value zero. Only input statements and I, F, E, D and G editing are affected.

BZ causes all blanks except leading blanks in numeric input to be replaced by zeros. Only input statements and I, F, E, D and G editing are affected.

H Format Control – Hollerith Control

The H format control writes the n characters following the H in the format specification and cannot be used on input.

The basic form of this format specification is:

```
nHc1cn...
```

where n is the number of characters to print and c1 through cn are the characters to print.

O Format Control Octal Values

The O and Z field descriptors transfer octal or hexadecimal values and can be used with an integer data type. They have the form:

```
Ow[.m] and Zw[.m]
```

where w specifies the field width and m indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) octal or hexadecimal characters only. An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the O and Z field descriptors transfer the octal and hexadecimal values, respectively, of the corresponding I/O list element, right-justified, to an external field that is w characters long.

- If the value to be transmitted does not fill the field, leading spaces are inserted.
- If the value is too large for the field, the entire field is filled with asterisks.
- If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary.
- If m is zero, and the internal representation is zero, the external field is blank-filled.

P Format Specifier – Scale Control

The P format specifier is the scale factor format.

```
kP
```

This specifier is applied as follows.

- With F, E, D and G editing on input and F editing on output, the external number equals the internal number multiplied by 10**k .
- If there is an exponent in the field on input, editing with F, E, D and G the scale factor has no effect.
- On output with E and D editing, the basic real constant part of the number is multiplied by 10**k and the exponent reduced by k.
- On output with G editing, the effect of the scale factor is suspended unless the size of the datum to be edited is outside the range permitted for F editing.
- On output if E editing is required, the scale factor has the same effect as with E output editing.

The following example uses a scale factor.

```
 DIMENSION
A(6)
 DO 10 I = 1,6
10 A(I) = 25.
 TYPE 100,A
100 FORMAT(' ',F8.2,2PF8.2,F8.2)
```

This example produces this output:

```
25.00
2500.00 2500.00 2500.00 2500.00 2500.00
```

Note

The effect of the scale factor continues until another scale factor is used.

Q Format Control - Quantity

The Q edit descriptor calculates the number of characters remaining in the input record and stores that value in the next I/O list item. On output, the Q descriptor skips the next I/O item.

r Format Control - Rounding

The rounding edit descriptors are valid in format processing, such as in a READ or WRITE statement. The specific rounding mode takes effect immediately when encountered, and stays in effect until either another descriptor is encountered or until the end of the READ and WRITE statement. Table 5.3 lists the edit descriptors associated with rounding.

Table 5.3. Format Character Controls for Rounding Printer

This descriptor ...	... Indicates this type of rounding
rc	round compatible
rd	round down
rn	round nearest
rp	round as processor_defined
ru	round up
rz	round zero

Both `nearest` and `compatible` refer to closest representable value. If these are equidistant, then the rounding is processor-dependent for `nearest` and the value away from zero for `compatible`.

S Format Control – Sign Control

The S format specifier restores the default processing for writing a plus; the default is SS processing.

SP forces the processor to write a plus in any position where an optional plus is found in numeric output fields, this only affects output statements.

SS stops the processor from writing a plus in any position where an optional plus is found in numeric output fields, this only affects output statements.

T, TL and X Format Controls – Spaces and Tab Controls

The T specifier controls which portion of a record in an iolist value is read from or written to a file. The general form, which specifies that the nth value is to be written to or from a record, is as follows:

```
Tn
```

The TL form specifies the relative position to the left of the data to be read or written, and specifies that the nth character to the left of the current position is to be written to or from the record. If the current position is less than or equal to n, the transmission will begin at position one of the record.

```
TLn
```

The TR form specifies the relative position to the right of the data to be read or written, and specifies that the nth character to the right of the current position is to be written to or from the record.

```
TRn
```

The X control specifies a number of characters to skip forward, and that the next character to be written to or from is n characters forward from the current position.

```
nX
```

The following example uses the X format specifier:

```
 NPAGE = 19
 WRITE ( 6, 90) NPAGE
90 FORMAT('1PAGE NUMBER ,I2, 16X, 'SALES REPORT, Cont.')
```

produces:

```
PAGE NUMBER 19 SALES REPORT, Cont.
```

The following example shows use of the T format specifier:

```
 PRINT 25
25 FORMAT (T41,'COLUMN 2',T21,'COLUMN 1')
```

produces:

```
 COLUMN 1 COLUMN 2
```

Z Format Control Hexadecimal Values

The O and Z field descriptors transfer octal or hexadecimal values and can be used with any integer data type. They have the form:

```
Ow[.m] and Zw[.m]
```

where w specifies the field width and m indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) octal or hexadecimal characters only. An all-blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the O and Z field descriptors transfer the octal and hexadecimal values, respectively, of the corresponding I/O list element, right-justified, to an external field that is w characters long.

- If the value to be transmitted does not fill the field, leading spaces are inserted.
- If the value is too large for the field, the entire field is filled with asterisks.
- If m is present, the external field consists of at least m digits, and is zero-filled on the left if necessary.
- If m is zero, and the internal representation is zero, the external field is blank-filled.

Slash Format Control / – End of Record

The slash (/) control indicates the end of data transfer on the current record.

On input from a file connected for sequential access, the rest of the current record is skipped and the file positioned at the start of the next record.

On output a new record is created which becomes the last and current record.

- For an internal file connected for direct access, the record is filled with blank characters.
- For a direct access file, the record number is increased by one and the file is positioned at the start of the record.

Note

Multiple slashes are permitted, thus multiple records are skipped.

The : Format Specifier – Format Termination

The (:) control terminates format control if there are no more items in the input/output list. It has no effect if there are any items left in the list.

$ Format Control

The $ field descriptor allows the programmer to control carriage control conventions on output. It is ignored on input. For example, on terminal output, it can be used for prompting.

The form of the $ field descriptor is:

```
$
```

Variable Format Expressions

Variable format expressions, <expr>, are supported in pgf77 extension only. They provide a means for substituting runtime expressions for the field width, other parameters for the field and edit descriptors in a FORMAT statement (except for the H field descriptor and repeat counts).

Variable format expressions are enclosed in "<" and ">" and are evaluated each time they are encountered in the scan of a format. If the value of a variable used in the expression changes during the execution of the I/O statement, the new value is used the next time the format item containing the expression is processed.

Non-advancing Input and Output

Non-advancing input/output is character-oriented and applies to sequential access formatted external files. The file position is after the last character read or written and not automatically advanced to the next record.

For non-advancing input/output, use the ADVANCE='NO' specifier. Two other specifiers apply to non-advancing IO: EOR applies when end of record is detected and SIZE returns the number of characters read.

List-directed formatting

List-directed formatting is an abbreviated form of input/output that does not require the use of a format specification. The type of the data determines how a value is read/written. On output, it is not always accurate enough for certain ranges of values. The characters in a list-directed record constitute a sequence of values which cannot contain embedded blanks except those permitted within a character string.

To use list-directed input/output formatting, specify a * for the list of format requirements, as illustrated in the following example that uses list-directed output:

```
READ( 1, * ) VAL1, VAL2
```

List-directed input

The form of the value being input must be acceptable for the type of item in the iolist. Blanks must not be used as zeros nor be embedded in constants except in a character constant or within a type complex form contained in parentheses.

Table 5.4. List Directed Input Values

Input List Type	Form
Integer	A numeric input field.
Real	A numeric input field suitable for F editing with no fractional part unless a decimal point is used.
Double precision	Same as for real.
Complex	An ordered pair of numbers contained within parentheses as shown: (real part, imaginary part).
Logical	A logical field without any slashes or commas.
Character	A non-empty character string within apostrophes. A character constant can be continued on as many records as required. Blanks, slashes and commas can be used.

A null value has no effect on the definition status of the corresponding iolist item. A null value cannot represent just one part of a complex constant but may represent the entire complex constant. A slash encountered as a value separator stops the execution of that input statement after the assignment of the previous value. If there are further items in the list, they are treated as if they are null values.

Commas may be used to separate the input values. If there are consecutive commas, or if the first non-blank character of a record is a comma, the input value is a null value. Input values may also be repeated.

In the following example of list-directed formatting, assume that A and K are defined as follows and all other variables are undefined.

```
A= -1.5
K= 125
```

Suppose that you have an input file the contains the following record, where the / terminates the input and consecutive commas indicate a null:

```
10,-14,25.2,-76,313,,29/
```

Further suppose that you use the following statement to read in the list from the input file:

```
READ * I, J, X, Y, Z, A, C, K
```

The variables are assigned the following values by the list-directed input/output mechanism:

I=10	J=-14	X=25.2	Y=-76.0
Z=313.0	A=-1.5	C=29	K=125

In the example the value for A does not change because the input record is null.Input is terminated with the / so no input is read for K, so the program assumes null and K retains its previous value.

List-directed output

List directed input/output is an abbreviated form of formatted input/output that does not require the use of a format specification. Depending on the type of the data item or data items in the iolist, data is transferred to

or from the file, using a default, and not necessarily accurate format specification. The data type of each item appearing in the iolist is formatted according to the rules in the following table:

Table 5.5. Default List Directed Output Formatting

Data Type	Default Formatting
BYTE	I5
INTEGER*2	I7
INTEGER*4	I12
INTEGER*8	I24
LOGICAL*1	I5 (L2[a])
LOGICAL*2	L2
LOGICAL*4	L2
LOGICAL*8	L2
REAL*4	G15.7e2
REAL*8	G25.16e3
COMPLEX*8	(G15.7e2, G15.7e2)
COMPLEX*16	(G25.16e3, G25.16e3)
CHAR *n	An

[a]This format is applied when the option –Munixlogical is selected when compiling.

The length of a record is less than 80 characters; if the output of an item would cause the length to exceed 80 characters, a new record is created.

The following rules and guidelines apply when using list-directed output:

- New records may begin as necessary.
- Logical output constants are T for true and F for false.
- Complex constants are contained within parentheses with the real and imaginary parts separated by a comma.
- Character constants are not delimited by apostrophes and have each internal apostrophe (if any are present) represented externally by one apostrophe.
- Each output record begins with a blank character to provide carriage control when the record is printed.
- A typeless value output with list-directed I/O is output in hexadecimal form by default. There is no other octal or hexadecimal capability with list-directed I/O.

Commas in External Field

Use of the comma in an external field eliminates the need to "count spaces" to have data match format edit descriptors. The use of a comma to terminate an input field and thus avoid padding the field is fully supported.

Character Encoding Format

Users can specify input/output encoding using the `encoding= specifier` on the OPEN statement. Further, the use of this specifier with the INQUIRE statement returns the encoding of the file:

`UTF-8` specifies the file is connected for UTF-8 I/O or that the processor can detect this format in some way.
`UNKNOWN` specifies the processor cannot detect the format.
A processor-dependent value indicates the file is in another known format, such as `UTF-16LE`.

Namelist Groups

The NAMELIST statement allows for the definition of namelist groups. A namelist group allows for a special type of formatted input/output, where data is transferred between a named group of data items defined in a NAMELIST statement and one or more records in a file.

The general form of a namelist statement is:

```
NAMELIST /group-name/ namelist [[,] /group-name/ namelist ]...
```

where:

group-name
: is the name of the namelist group.

namelist
: is the list of variables in the namelist group.

Namelist Input

Namelist input is accomplished using a READ statement by specifying a namelist group as the input item. The following statement shows the format:

```
READ ([unit=] u, [NML=] namelist-group [,control-information])
```

One or more records are processed which define the input for items in the namelist group.

The records are logically viewed as follows:

```
$group-name item=value [,item=value].... $ [END]
```

The following rules describe these input records:

- The start or end delimiter ($) may be an ampersand (&).
- The start delimiter must begin in column 2 of a record.
- The group-name begins immediately after the start delimiter.
- The spaces or tabs may not appear within the group-name, within any item, or within any constants.
- The value may be constants as are allowed for list directed input, or they may be a list of constants separated by commas (,). A list of items is used to assign consecutive values to consecutive elements of an array.

- Spaces or tabs may precede the item, the = and the constants.
- Array items may be subscripted.
- Character items may have substrings.

Namelist Output

Namelist output is accomplished using a READ statement by specifying a namelist group as the output item. The following statement shows the format:

```
WRITE ([unit=] u, [NML=] namelist-group [,control-information])
```

The records output are logically viewed as follows:

```
$group-name
item = value
$ [END]
```

The following rules describe these output records:

- One record is output per value.
- Multiple values are separated by a comma (,).
- Values are formatted according to the rules of the list-directed write. Exception: character items are delimited by an apostrophe (').
- An apostrophe (') or a quote (") in the value is represented by two consecutive apostrophes or quotes.

Recursive Input/Output

Recursive Input/Output allows you to execute an input/output statement while another input/output statement is being execution. This capability is available under these conditions:

- External files, such as a child data transfer statement invoking derived type input/output
- Internal files, such as input/output to/from an internal file where that statement does not modify any internal file other than its own.

Input and Output of IEEE Infinities and NaNs

In Fortran 2003, input and output of IEEE infinities and NaNs is specified.

All edit descriptors for reals treat these values in the same way; only the field width is required.

Output Format

Output for infinities and NaNs is right-justified within the output field. For list-directed output the output field is the minimum size to hold the result. The format is this:

For minus infinity

```
-Infinity
-Inf
```

For plus infinity	`Infinity`
	`Inf`
	`+Infinity`
	`+Inf`

For a Nan	NaN, optionally followed by non-blank characters in parenthesis

Input Format

Input for infinities and NaNs is similar to the output except that case is not significant.

The format is this:

For minus infinity	`-Infinity` `-Inf`

For plus infinity	`Infinity`
	`Inf`
	`+Infinity`
	`+Inf`

For a Nan	NaN, optionally followed by non-blank characters in parenthesis When no non-blank character is present, the NaN is a quiet NaN.

Chapter 6. Fortran Intrinsics

An intrinsic is a function available in a given language whose implementation is handled specifically by the compiler. Typically, an intrinsic substitutes a sequence of automatically-generated instructions for the original function call. Since the compiler has an intimate knowledge of the intrinsic function, it can better integrate it and optimize it for the situation.

This chapter lists the FORTRAN 77 and Fortran 90/95 intrinsics and subroutines and Fortran 2003 intrinsic modules. The Fortran processor, rather than the user or a third party, provides the intrinsic functions and intrinsic modules.

For details on the standard intrinsics, refer to the Fortran language specifications readily available on the internet. The `Origin` column in the tables in this chapter provides the Fortran language origin of the statement; for example, F95 indicates the statement is from Fortran 95.

Intrinsics Support

The tables in this section contain the FORTRAN 77, Fortran 90/95 and Fortran 2003 intrinsics that are supported. At the top of each reference page is a brief description of the statement followed by a header that indicates the origin of the statement. The following list describes the meaning of the origin abbreviations.

F77
: FORTRAN 77 intrinsics that are essentially unchanged from the original FORTRAN 77 standard and are supported by the PGF77 compiler.

F77 extension
: The statement is an extension to the Fortran language.

F90/F95
: The statement is either new for Fortran 90/95 or significantly changed in Fortran 95 from its original FORTRAN 77 definition and is supported by the PGF95 and PGFORTRAN compilers.

F2003
: The statement is new for Fortran 2003.

The functions in the following table are specific to Fortran 90/95 unless otherwise specified.

Table 6.1. Fortran 90/95 Bit Manipulation Functions and Subroutines

Generic Name	Purpose	Num. Args	Argument Type	Result Type
AND	Performs a logical AND on corresponding bits of the arguments.	2	ANY type except CHAR or COMPLEX	
BIT_SIZE	Return the number of bits (the precision) of the integer argument.	1	INTEGER	INTEGER
BTEST	Tests the binary value of a bit in a specified position of an integer argument.	2	INTEGER, INTEGER	LOGICAL
IAND	Perform a bit-by-bit logical AND on the arguments.	2	INTEGER, INTEGER (of same kind)	INTEGER
IBCLR	Clears one bit to zero.	2	INTEGER, INTEGER >=0	INTEGER
IBITS	Extracts a sequence of bits.		INTEGER, INTEGER >=0, INTEGER >=0	INTEGER
IBSET	Sets one bit to one.	2	INTEGER, INTEGER >=0	INTEGER
IEOR	Perform a bit-by-bit logical exclusive OR on the arguments.	2	INTEGER, INTEGER (of same kind)	INTEGER
IOR	Perform a bit-by-bit logical OR on the arguments.	2	INTEGER, INTEGER (of same kind)	INTEGER
ISHFT	Perform a logical shift.	2	INTEGER, INTEGER	INTEGER
ISHFTC	Perform a circular shift of the rightmost bits.	2 or 3	INTEGER, INTEGER INTEGER, INTEGER, INTEGER	INTEGER
LSHIFT	Perform a logical shift to the left.	2	INTEGER, INTEGER	INTEGER
MVBITS	Copies bit sequence	5	INTEGER(IN), INTEGER(IN), INTEGER(IN), INTEGER(INOUT), INTEGER(IN)	none
NOT	Perform a bit-by-bit logical complement on the argument.	2	INTEGER	INTEGER
OR	Performs a logical OR on each bit of the arguments.	2	ANY type except CHAR or COMPLEX	
POPCNT (F2008)	Return the number of one bits.	1	INTEGER or bits	INTEGER
POPPAR (F2008)	Return the bitwise parity.	1	INTEGER or bits	INTEGER
RSHIFT	Perform a logical shift to the right.	2	INTEGER, INTEGER	INTEGER

Generic Name	Purpose	Num. Args	Argument Type	Result Type
SHIFT	Perform a logical shift.	2	Any type except CHAR or COMPLEX, INTEGER	
XOR	Performs a logical exclusive OR on each bit of the arguments.	2	INTEGER, INTEGER	INTEGER
ZEXT	Zero-extend the argument.	1	INTEGER or LOGICAL	INTEGER

The functions in the following table are specific to Fortran 90/95 unless otherwise specified.

Table 6.2. Elemental Character and Logical Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
ACHAR	Return character in specified ASCII collating position.	1	INTEGER	CHARACTER
ADJUSTL	Left adjust string	1	CHARACTER	CHARACTER
ADJUSTR	Right adjust string	1	CHARACTER	CHARACTER
CHAR (f77)	Return character with specified ASCII value.	1	LOGICAL*1 INTEGER	CHARACTER CHARACTER
IACHAR	Return position of character in ASCII collating sequence.	1	CHARACTER	INTEGER
ICHAR	Return position of character in the character set's collating sequence.	1	CHARACTER	INTEGER
INDEX	Return starting position of substring within first string.	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER INTEGER
LEN	Returns the length of string	1	CHARACTER	INTEGER
LEN_TRIM	Returns the length of the supplied string minus the number of trailing blanks.	1	CHARACTER	INTEGER
LGE	Test the supplied strings to determine if the first string is lexically greater than or equal to the second.	2	CHARACTER, CHARACTER	LOGICAL
LGT	Test the supplied strings to determine if the first string is lexically greater than the second.	2	CHARACTER, CHARACTER	LOGICAL

Generic Name	Purpose	Num. Args	Argument Type	Result Type
LLE	Test the supplied strings to determine if the first string is lexically less than or equal to the second.	2	CHARACTER, CHARACTER	LOGICAL
LLT	Test the supplied strings to determine if the first string is lexically less than the second.	2	CHARACTER, CHARACTER	LOGICAL
LOGICAL	Logical conversion	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL
SCAN	Scan string for characters in set	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER INTEGER
VERIFY	Determine if string contains all characters in set	2 3	CHARACTER, CHARACTER CHARACTER, CHARACTER, LOGICAL	INTEGER

Table 6.3. Fortran 90/95 Vector/Matrix Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
DOT_PRODUCT	Perform dot product on two vectors	2	NONCHAR*K, NONCHAR*K	NONCHAR*K
MATMUL	Perform matrix multiply on two matrices	2	NONCHAR*K, NONCHAR*K	NONCHAR*K

Table 6.4. Fortran 90/95 Array Reduction Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
ALL	Determine if all array values are true	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL
ANY	Determine if any array value is true	1 2	LOGICAL LOGICAL, INTEGER	LOGICAL LOGICAL
COUNT	Count true values in array	1 2	LOGICAL LOGICAL, INTEGER	INTEGER INTEGER

Generic Name	Purpose	Num. Args	Argument Type	Result Type
MAXLOC	Determine position of array element with maximum value	1 2 2 3 1 2 2 3	INTEGER INTEGER, LOGICAL INTEGER, INTEGER INTEGER, INTEGER, LOGICAL REAL REAL, LOGICAL REAL, INTEGER REAL, INTEGER, LOGICAL	INTEGER
MAXVAL	Determine maximum value of array elements	1 2 2 3 1 2 2 3	INTEGER INTEGER, LOGICAL INTEGER, INTEGER INTEGER, INTEGER, LOGICAL REAL REAL, LOGICAL REAL, INTEGER REAL, INTEGER, LOGICAL	INTEGER INTEGER INTEGER INTEGER REAL REAL REAL REAL
MINLOC	Determine position of array element with minimum value	1 2 2 3 1 2 2 3	INTEGER INTEGER, LOGICAL INTEGER, INTEGER INTEGER, INTEGER, LOGICAL REAL REAL, LOGICAL REAL, INTEGER REAL, INTEGER, LOGICAL	INTEGER
MINVAL	Determine minimum value of array elements	1 2 2 3 1 2 2 3	INTEGER INTEGER, LOGICAL INTEGER, INTEGER INTEGER, INTEGER, LOGICAL REAL REAL, LOGICAL REAL, INTEGER REAL, INTEGER, LOGICAL	INTEGER INTEGER INTEGER INTEGER REAL REAL REAL REAL
PRODUCT	Calculate the product of the elements of an array	1 2 2 3	NUMERIC NUMERIC, LOGICAL NUMERIC, INTEGER NUMERIC, INTEGER, LOGICAL	NUMERIC

Generic Name	Purpose	Num. Args	Argument Type	Result Type
SUM	Calculate the sum of the elements of an array	1 2 2 3	NUMERIC NUMERIC, LOGICAL NUMERIC, INTEGER NUMERIC, INTEGER, LOGICAL	NUMERIC

Table 6.5. Fortran 90/95 String Construction Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
REPEAT	Concatenate copies of a string	2	CHARACTER, INTEGER	CHARACTER
TRIM	Remove trailing blanks from a string	1	CHARACTER	CHARACTER

Table 6.6. Fortran 90/95 Array Construction/Manipulation Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
CSHIFT	Perform circular shift on array	2 3	ARRAY, INTEGER ARRAY, INTEGER, INTEGER	ARRAY[a] ARRAY[a]
EOSHIFT	Perform end-off shift on array	2 3 3 4	ARRAY, INTEGER ARRAY, INTEGER, any[a] ARRAY, INTEGER, INTEGER ARRAY, INTEGER, any[a], INTEGER	ARRAY[a] ARRAY[a] ARRAY[a] ARRAY[a]
MERGE	Merge two arguments based on logical mask	3	any, any[a], LOGICAL	any[a]
PACK	Pack array into rank-one array	2 3	ARRAY, LOGICAL ARRAY, LOGICAL, VECTOR[a]	ARRAY[a]
RESHAPE	Change the shape of an array	2 3 3 4	ARRAY, INTEGER ARRAY, INTEGER, ARRAY[a] ARRAY, INTEGER, INTEGER ARRAY, INTEGER, ARRAY[a], INTEGER	ARRAY[a]
SPREAD	Replicates an array by adding a dimension	3	any, INTEGER, INTEGER	ARRAY[a]
TRANSPOSE	Transpose an array of rank two	1	ARRAY	ARRAY[a]

Generic Name	Purpose	Num. Args	Argument Type	Result Type
UNPACK	Unpack a rank-one array into an array of multiple dimensions	3	VECTOR, LOGICAL, ARRAY[a]	ARRAY[a]

[a]Must be of the same type as the first argument.

Table 6.7. Fortran 90/95 General Inquiry Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
ASSOCIATED	Determine association status	12	POINTER, POINTER,..., POINTER, TARGET	LOGICAL LOGICAL
KIND	Determine argument's kind	1	any intrinsic type	INTEGER
PRESENT	Determine presence of optional argument	1	any	LOGICAL

Table 6.8. Fortran 90/95 Numeric Inquiry Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
DIGITS	Determine number of significant digits	1 1	INTEGER REAL	INTEGER
EPSILON	Smallest representable number	1	REAL	REAL
HUGE	Largest representable number	1 1	INTEGER REAL	INTEGER REAL
MAXEXPONENT	Value of maximum exponent	1	REAL	INTEGER
MINEXPONENT	Value of minimum exponent	1	REAL	INTEGER
PRECISION	Decimal precision	1 1	REAL COMPLEX	INTEGER INTEGER
RADIX	Base of model	1 1	INTEGER REAL	INTEGER INTEGER
RANGE				

Generic Name	Purpose	Number of Args	Argument Type	Result Type
	Decimal exponent range	1	INTEGER	INTEGER
		1	REAL	INTEGER
		1	COMPLEX	INTEGER
SELECTED_INT_KIND	Kind type titlemeter in range	1	INTEGER	INTEGER
SELECTED_REAL_KIND	Kind type titlemeter in range	1	INTEGER	INTEGER
		2	INTEGER, INTEGER	INTEGER
TINY	Smallest representable positive number	1	REAL	REAL

Table 6.9. Fortran 90/95 Array Inquiry Functions

Generic Name	Purpose	Number of Args	Argument Type	Result Type
ALLOCATED	Determine if array is allocated	1	ARRAY	LOGICAL
LBOUND	Determine lower bounds	1	ARRAY	INTEGER
		2	ARRAY, INTEGER	
SHAPE	Determine shape	1	any	INTEGER
SIZE	Determine number of elements	1	ARRAY	INTEGER
		2	ARRAY, INTEGER	
UBOUND	Determine upper bounds	1	ARRAY	INTEGER
		2	ARRAY, INTEGER	

Table 6.10. Fortran 90/95 Subroutines

Generic Name	Purpose	Number of Args	Argument Type
CPU_TIME	Returns processor time	1	REAL (OUT)

Generic Name	Purpose	Number of Args	Argument Type
DATE_AND_TIME	Returns date and time	4 (optional)	DATE (CHARACTER, OUT) TIME (CHARACTER, OUT) ZONE (CHARACTER, OUT) VALUES (INTEGER, OUT)
RANDOM_NUMBER	Generate pseudo-random numbers	1	REAL (OUT)
RANDOM_SEED	Set or query pseudo-random number generator	0 1 1 1	 SIZE (INTEGER, OUT) PUT (INTEGER ARRAY, IN) GET (INTEGER ARRAY, OUT)
SYSTEM_CLOCK	Query real time clock	3 (optional)	COUNT (INTEGER, OUT) COUNT_RATE (REAL, OUT) COUNT_MAX (INTEGER, OUT)

Table 6.11. Fortran 90/95 Transfer Function

Generic Name	Purpose	Number of Args	Argument Type	Result Type
TRANSFER	Change type but maintain bit representation	2 3	any, any any, any, INTEGER	any[a]

[a]Must be of the same type as the second argument.

Table 6.12. Arithmetic Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
ABS	Return absolute value of the supplied argument.	1	INTEGER REAL COMPLEX	INTEGER REAL COMPLEX
ACOS	Return the arccosine (in radians) of the specified value	1	REAL	REAL
ACOSD	Return the arccosine (in degrees) of the specified value	1	REAL	REAL
AIMAG	Return the value of the imaginary part of a complex number.	1	COMPLEX	REAL

Generic Name	Purpose	Num. Args	Argument Type	Result Type
AINT	Truncate the supplied value to a whole number.	2	REAL, INTEGER	REAL
AND	Performs a logical AND on corresponding bits of the arguments.	2	ANY type except CHAR or COMPLEX	
ANINT	Return the nearest whole number to the supplied argument.	2	REAL, INTEGER	REAL
ASIN	Return the arcsine (in radians) of the specified value	1	REAL	REAL
ASIND	Return the arcsine (in degrees) of the specified value	1	REAL	REAL
ATAN	Return the arctangent (in radians) of the specified value	1	REAL	REAL
ATAN2	Return the arctangent (in radians) of the specified pair of values.	2	REAL, REAL	REAL
ATAN2D	Return the arctangent (in degrees) of the specified pair of values	1	REAL, REAL	REAL
ATAND	Return the arctangent (in degrees) of the specified value	1	REAL	REAL
CEILING	Return the least integer greater than or equal to the supplied real argument.	2	REAL, KIND	INTEGER
CMPLX	Convert the supplied argument or arguments to complex type.	2 3	INTEGER, REAL, or COMPLEX; INTEGER, REAL, or COMPLEX; INTEGER, REAL, or COMPLEX; INTEGER or REAL KIND	COMPLEX
COMPL	Performs a logical complement on the argument.	1	ANY, except CHAR or COMPLEX	
COS	Return the cosine (in radians) of the specified value	1	REAL COMPLEX	REAL
COSD	Return the cosine (in degrees) of the specified value	1	REAL COMPLEX	REAL

Generic Name	Purpose	Num. Args	Argument Type	Result Type
COSH	Return the hyperbolic cosine of the specified value	1	REAL	REAL
DBLE	Convert to double precision real.		INTEGER, REAL, or COMPLEX	REAL
DCMPLX	Convert the supplied argument or arguments to double complex type.	1 2	INTEGER, REAL, or COMPLEX INTEGER, REAL	DOUBLE COMPLEX
DPROD	Double precision real product.	2	REAL, REAL	REAL (double precision)
EQV	Performs a logical exclusive NOR on the arguments.	2	ANY except CHAR or COMPLEX	
EXP	Exponential function.	1	REAL COMPLEX	REAL COMPLEX
EXPONENT	Return the exponent part of a real number.	1	REAL	INTEGER
FLOOR	Return the greatest integer less than or equal to the supplied real argument.	1 2	REAL REAL, KIND	REAL KIND
FRACTION	Return the fractional part of a real number.	1	REAL	INTEGER
IINT	Converts a value to a short integer type.	1	INTEGER, REAL, or COMPLEX	INTEGER
ININT	Returns the nearest short integer to the real argument.	1	REAL	INTEGER
INT	Converts a value to integer type.	1 2	INTEGER, REAL, or COMPLEX INTEGER, REAL, or COMPLEX; KIND	INTEGER
INT8	Converts a real value to a long integer type.	1	REAL	INTEGER
IZEXT	Zero-extend the argument.	1	LOGICAL or INTEGER	INTEGER
JINT	Converts a value to an integer type.	1	INTEGER, REAL, or COMPLEX	INTEGER
JNINT	Returns the nearest integer to the real argument.	1	REAL	INTEGER
KNINT	Returns the nearest integer to the real argument.	1	REAL	INTEGER (long)

Generic Name	Purpose	Num. Args	Argument Type	Result Type
LOG	Returns the natural logarithm.	1	REAL or COMPLEX	REAL
LOG10	Returns the common logarithm.	1	REAL	REAL
MAX	Return the maximum value of the supplied arguments.	2 or more	INTEGER or REAL (all of same kind)	Same as Argument Type
MIN	Return the minimum value of the supplied arguments.	2 or more	INTEGER or REAL (all of same kind)	Same as Argument Type
MOD	Find the remainder.	2 or more	INTEGER or REAL, INTEGER or REAL (all of same kind)	Same as Argument Type
MODULO	Return the modulo value of the arguments.	2 or more	INTEGER or REAL, INTEGER or REAL (all of same kind)	Same as Argument Type
NEAREST	Returns the nearest different machine representable number in a given direction.	2	REAL, non-zero REAL	REAL
NEQV	Performs a logical exclusive OR on the arguments.	2	ANY except CHAR or COMPLEX	
NINT	Converts a value to integer type.	1 2	REAL REAL, KIND	INTEGER
REAL	Convert the argument to real.	1 2	IINTEGER, REAL, or COMPLEX INTEGER, REAL, or COMPLEX; KIND	REAL
RRSPACING	Return the reciprocal of the relative spacing of model numbers near the argument value.	1	REAL	REAL
SET_ EXPONENT	Returns the model number whose fractional part is the fractional part of the model representation of the first argument and whose exponent part is the second argument.	2	REAL, INTEGER	REAL
SIGN	Return the absolute value of A times the sign of B.	2	INTEGER or REAL, INTEGER or REAL (of same kind)	Same as Argument
SIN	Return the sine (in radians) of the specified value	1	REAL COMPLEX	REAL

Generic Name	Purpose	Num. Args	Argument Type	Result Type
SIND	Return the sine (in degrees) of the specified value	1	REAL COMPLEX	REAL
SINH	Return the hyperbolic sine of the specified value	1	REAL	REAL
SPACING	Return the relative spacing of model numbers near the argument value.	1	REAL	REAL
SQRT	Return the square root of the argument.	1	REAL COMPLEX	REAL COMPLEX
TAN	Return the tangent (in radians) of the specified value	1	REAL	REAL
TAND	Return the tangent (in degrees) of the specified value	1	REAL	REAL
TANH	Return the hyperbolic tangent of the specified value	1	REAL	REAL

Table 6.13. Fortran 2003 and 2008 Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
COMMAND_ ARGUMENT_COUNT	Returns a scalar of type default integer that is equal to the number of arguments passed on the command line when the containing program was invoked. If there were no command arguments passed, the result is 0.	0		INTEGER
EXTENDS_TYPE_OF	Determines whether the dynamic type of A is an extension type of the dynamic type of B.	2	Objects of extensible type	LOGICAL SCALAR
GET_COMMAND_ ARGUMENT	Returns the specified command line argument of the command that invoked the program.	1 to 4	INTEGER plus optinally: CHAR, INTEGER, INTEGER	A command argument

Generic Name	Purpose	Num. Args	Argument Type	Result Type
GET_COMMAND	Returns the entire command line that was used to invoke the program.	0 to 3	CHAR, INTEGER, INTEGER	A command line
GET_ENVIRONMENT _VARIABLE	Returns the value of the specified environment variable.	1 to 5	CHAR, CHAR, INTEGER, INTEGER, LOGICAL	
IS_IOSTAT_END	Test whether a variable has the value of the I/O status: “end of file”.	1	INTEGER	LOGICAL
IS_IOSTAT_EOR	Test whether a variable has the value of the I/O status: “end of record”.	1	INTEGER	LOGICAL
LEADZ (F2008)	Counts the number of leading zero bits.	1	INTEGER or bits	INTEGER
MOVE_ALLOC	Moves an allocation from one allocatable object to another.	2	Any type and rank	none
NEW_LINE	Return the newline character.	1	CHARACTER	CHARACTER
SAME_TYPE_AS	Determines whether the dynamic type of A is the same as the dynamic type of B.	2	Objects of extensible type	LOGICAL SCALAR
SCALE	Return the value X * bi where b is the base of the number system in use for X.	2	REAL, INTEGER	REAL

Table 6.14. Miscellaneous Functions

Generic Name	Purpose	Num. Args	Argument Type	Result Type
LOC	Return address of argument	1	NUMERIC	INTEGER
NULL	Assign disassociated status	0		POINTER
		1	POINTER	POINTER

ACOSD

Return the arccosine (in degrees) of the specified value.

F77

Synopsis

```
ACOSD(X)
```

Arguments

The argument X must be a real value.

Return Value

The real value representing the arccosine in degrees.

AND

Performs a logical AND on corresponding bits of the arguments.

F77 extension

Synopsis

```
AND(M, N)
```

Arguments

The arguments M and N may be of any type except for character and complex.

Return Value

The return value is typeless.

ASIND

Return the arcsine (in degrees) of the specified value.

F77

Synopsis

```
ASIND(X)
```

Argument

The argument X must be of type real and have absolute value <= 1.

Return Value

The real value representing the arcsine in degrees.

ASSOCIATED

Determines the association status of the supplied argument or determines if the supplied pointer is associated with the supplied target.

F90

Synopsis

```
ASSOCIATED(POINTER [,TARGET])
```

Arguments

The POINTER argument is a pointer of any type. The optional argument TARGET is a pointer or a target. If it is a pointer it must not be undefined.

Return Value

If TARGET is not supplied the function returns logical true if POINTER is associated with a target and false otherwise.

If TARGET is present and is a target, then the function returns true if POINTER is associated with TARGET and false otherwise.

If TARGET is present and is a pointer, then the function returns true if POINTER and TARGET are associated with the same target and false otherwise.

ATAN2D

Return the arctangent (in degrees) of the specified value.

F77

Synopsis

```
ATAN2D(Y, X)
```

Arguments

The arguments X and Y must be of type real.

Return Value

A real number that is the arctangent for pairs of reals, X and Y, expressed in degrees. The result is the principal value of the nonzero complex number (X,Y).

ATAND

Return the arctangent (in degrees) of the specified value.

F77

Synopsis

```
ATAND(X)
```

Argument

The argument X must be of type real.

Return Value

The real value representing the arctangent in degrees.

COMPL

Performs a logical complement on the argument.

F77 extension

Synopsis

```
COMPL(M)
```

Arguments

The argument M may be of any type except for character and complex.

Return Value

The return value is typeless.

CONJG

Return the conjugate of the supplied complex number.

F77

Synopsis

```
CONJG(Z)
```

Argument

The argument Z is a complex number.

Return Value

A value of the same type and kind as the argument.

COSD

Return the cosine (in degrees) of the specified value.

F77

Synopsis

```
COSD(X)
```

Argument

The argument X must be of type real or complex.

Return Value

A real value of the same kind as the argument. The return value for a real argument is in degrees, or if complex, the real part is a value in degrees.

DIM

Returns the difference X-Y if the value is positive, otherwise it returns 0.

F77

Synopsis

```
DIM(X, Y)
```

Arguments

X must be of type integer or real. Y must be of the same type and kind as X.

Return Value

The result is the same type and kind as X with the value X-Y if X>Y, otherwise zero.

ININT

Returns the nearest short integer to the real argument.

F77 extension

Synopsis

```
ININT(A)
```

Arguments

The argument A must be a real.

Return Value

A short integer with value (A + .5 * SIGN(A)).

INT8

Converts a real value to a long integer type.

F77 extension

Synopsis

```
INT8(A)
```

Arguments

The argument A is of type real.

Return Value

The long integer value of the supplied argument.

IZEXT

Zero-extend the argument.

F77 extension

Synopsis

```
IZEXT(A)
```

Arguments

The argument A is of type logical or integer.

Return Value

A zero-extended short integer of the argument.

JINT

Converts a value to an integer type.

F77 extension

Synopsis

```
JINT(A)
```

Arguments

The argument A is of type integer, real, or complex.

Return Value

The integer value of the supplied argument.

- For a real number, if the absolute value of the real is less than 1, the return value is 0.
- If the absolute value is greater than 1, the result is the largest integer that does not exceed the real value.
- If argument is a complex number, the return value is the result of applying the real conversion to the real part of the complex number.

JNINT

Returns the nearest integer to the real argument.

F77 extension

Synopsis

```
JNINT(A)
```

Arguments

The argument A must be a real.

Return Value

An integer with value (A + .5 * SIGN(A)).

KNINT

Returns the nearest integer to the real argument.

F77 extension

Synopsis

```
KNINT(A)
```

Arguments

The argument A must be a real.

Return Value

A long integer with value (A + .5 * SIGN(A)).

LEADZ

Counts the number of leading zero bits.

F2003

Synopsis

```
LEADZ(I)
```

Arguments

I is of type integer or bits.

Return Value

The result is one of the following:

- If all of the bits of I are zero: BIT SIZE (I).
- If at least one of the bits of I is not zero: BIT SIZE (I) - 1 - k.

  k is the position of the leftmost 1 bit in I.

Description

LEADZ is an elemental function that returns the number of leading zero bits.

Examples

The following example returns the value 2.

```
LEADZ (B'001101000')
```

The following example returns the value 31 if BIT SIZE (1) has the value 32.

```
LEADZ (1)
```

LSHIFT

Perform a logical shift to the left.

F77 extension

Synopsis

```
LSHIFT(I, SHIFT)
```

Arguments

I and SHIFT are integer values.

Return Value

A value of the same type and kind as the argument I. It is the value of the argument I logically shifted left by SHIFT bits.

OR

Performs a logical OR on each bit of the arguments.

F77 extension

Synopsis

```
OR(M, N)
```

Arguments

The arguments M and N may be of any type except for character and complex.

Return Value

The return value is typeless.

RSHIFT

Perform a logical shift to the right.

F77 extension

Synopsis

```
RSHIFT(I, SHIFT)
```

Arguments

I and SHIFT are integer values.

Return Value

A value of the same type and kind as the argument I. It is the value of the argument I logically shifted right by SHIFT bits.

SHIFT

Perform a logical shift.

F77 extension

Synopsis

```
RSHIFT(I, SHIFT)
```

Arguments

The argument I may be of any type except character or complex. The argument SHIFT is of type integer.

Return Value

The return value is typeless. If SHIFT is positive, the result is I logically shifted left by SHIFT bits. If SHIFT is negative, the result is I logically shifted right by SHIFT bits.

SIND

Return the value in degrees of the sine of the argument.

F77

Synopsis

```
SIND(X)
```

Argument

The argument X must be of type real or complex.

Return Value

A value that has the same type as X and is expressed in degrees.

TAND

Return the tangent of the specified value.

F77

Synopsis

```
TAND(X)
```

Argument

The argument X must be of type real and have absolute value <= 1.

Return Value

A real value of the same KIND as the argument.

XOR

Performs a logical exclusive OR on each bit of the arguments.

F77 extension

Synopsis

```
XOR(M, N)
```

Arguments

The arguments M and N must be of integer type.

Return Value

An integer.

ZEXT

Zero-extend the argument.

F77 extension

Synopsis

```
ZEXT(A)
```

Arguments

The argument A is of type logical or integer.

Return Value

An integer.

Intrinsic Modules

Like an intrinsic function, the Fortran processor provides the intrinsic module. It is possible for a program to use an intrinsic module and a user-defined module of the same name, though they cannot both be referenced from the same scope.

- To use a user-defined module rather than an intrinsic module, specify the keyword *non-intrinsic* in the USE statement:

```
USE, non-intrinsic :: iso_fortran_env
```

- To use an intrinsic module rather than a user-defined one, specify the keyword *intrinsic* in the USE statement:

```
USE, intrinsic :: iso_fortran_env
```

Note

If both a user-defined and intrinsic module of the same name are available and locatable by the compiler, a USE statement without either of the keywords previously described accesses the user-defined module. If the compiler cannot locate the user-defined module, it accessed the intrinsic module and does not issue a warning.

Module IEEE_ARITHMETIC

The `ieee_arithmetic` intrinsic module provides access to two derived types, named constants of these types, and a collection of generic procedures.

Note

This module behaves as if it contained a **use** statement for the module `ieee_exceptions`, so all the features of `ieee_exceptions` are included. For information of this module, refer to "Module IEEE_EXCEPTIONS," on page 117.

IEEE_ARITHMETIC Derived Types

The `ieee_arithmetic` intrinsic module provides access to these two derived types: ieee_class_type and ieee_round_type.

ieee_class_type
: Identifies a class of floating point values.

ieee_round_type
: Identifies a particular round mode.

For both of these types, the following are true:

- The components are private.
- The only operations defined are == and /=.
- The return value is of type default logical.

  If the operator is ==, for two values of one of the derived types, this operator returns true if the values are the same; false, otherwise.

  If the operator is /=, for two values of one of the derived types, this operator returns true if the values are different; false, otherwise.
- Intrinsic assignment is available.

Table 6.15 provides a quick overview of the values that each derived type can take.

Table 6.15. IEEE_ARITHMETIC Derived Types

This derived type...	Must have one of these values...
ieee_class_type	ieee_signaling_nan ieee_quiet_nan ieee_negative_inf ieee_negative_normal ieee_negative_denormal ieee_negative_zero ieee_positive_zero ieee_positive_denormal ieee_positive_normal ieee_positive_inf ieee_other_value (Fortran 2003 only)
ieee_round_type	ieee_nearest - ieee_to_zero ieee_up ieee_down ieee_other (for modes other than IEEE ones)

IEEE_ARITHMETIC Inquiry Functions

The `ieee_arithmetic` intrinsic module supports a number of inquiry functions. Table 6.16 provides a list and brief description of what it means if the inquiry function returns `.true.` In all cases, if the condition for returning `.true.` is not met, the function returns `.false.`.

Table 6.16. IEEE_ARITHMETIC Inquiry Functions

	Returns .true. if ...	
This inquiry function...	**When optional arg. x is absent**	**When optional arg. x is present**
ieee_support_datatype([x])	The processor supports IEEE arithmetic for all reals	The processor supports IEEE arithmetic for all reals of the same kind type parameter as the real argument `x`.
ieee_support_denormal([x])	The processor supports IEEE denomalized numbers for all reals	The processor supports IEEE denomalized numbers for reals of the same kind type parameter as the real argument `x`.
ieee_support_divide([x])	The processor supports divide with the accuracy specified by IEEE standard for all reals	The processor supports divide with the accuracy specified by IEEE standard for reals of the same kind type parameter as the real argument `x`.

	Returns .true. if ...	
This inquiry function...	**When optional arg. x is absent**	**When optional arg. x is present**
ieee_support_inf([x])	The processor supports the IEEE infinity facility for all reals	The processor supports the IEEE infinity facility for reals of the same kind type parameter as the real argument x.
ieee_support_nan([x])	The processor supports the IEEE Not-A-Number facility for all reals	The processor supports the IEEE Not-A-Number facility for reals of the same kind type parameter as the real argument x.
ieee_support_rounding (round_value[,x])	For a round_value of ieee_round_type, the processor supports the rounding mode numbers for all reals	For a round_value of ieee_round_type, the processor supports the rounding mode numbers for reals of the same kind type parameter as the real argument x.
ieee_support_sqrt([x])	The processor implements the IEEE square root for all reals	The processor implements the IEEE square root for reals of the same kind type parameter as the real argument x.
ieee_support_standard([x])	The processor supports all IEEE facilities for all reals	The processor supports all IEEE facilities for reals of the same kind type parameter as the real argument x.
ieee_support_underflow_ control ([x])	(Fortran 2003 only) The processor supports control of the underflow mode for all reals	(Fortran 2003 only) The processor supports control of the underflow mode for reals of the same kind type parameter as the real argument x.

IEEE_ARITHMETIC Elemental Functions

The `ieee_arithmetic` intrinsic module supports a number of elemental functions. Table 6.17 provides a list and brief description of the return value. In all cases involving a return value of true or false, if the condition for returning `.true.` is not met, the subroutine returns `.false.`.

Table 6.17. IEEE_ARITHMETIC Elemental Functions

This elemental function...	**Does this...**
ieee_class(x)	Returns the IEEE class of the real argument x.
ieee_copy_sign(x,y)	Returns a real with the same type parameter as the real argument x, holding the value of x with the sign of y.

This elemental function...	Does this...

ieee_is_finite(x)	Returns `.true.` if `ieee_class(x)` has one of these values:
	`ieee_negative_normal`
	`ieee_negative_denormal`
	`ieee_negative_zero`
	`ieee_positive_zero`
	`ieee_positive_denormal`
	`ieee_positive_normal`

ieee_is_nan(x)	Returns `.true.` if the value of `x` is an IEEE NaN.

ieee_is_negative(x)	Returns `.true.` if `ieee_class(x)` as one of these values:
	`ieee_negative_normal`
	`ieee_negative_denormal`
	`ieee_negative_zero`
	`ieee_negative_inf`

ieee_is_normal(x)	Returns `.true.` if `ieee_class(x)` has one of these values:
	`ieee_negative_normal`
	`ieee_negative_zero`
	`ieee_positive_zero`
	`ieee_positive_normal`

ieee_is_logb(x)	Returns a real with the same type parameter as the argument `x`. If x is neither zero, infinity, nor NaN, the value of the result is the unbiased exponent of x: `exponent(x)-1.` If x is 0 and `ieee_support_inf(x)` is true, the result is -infinity. If x is 0 and `ieee_support_inf(x)` is not true, the result is `-huge(x)`.
ieee_next_after(x,y)	Returns a real with the same type parameter as the argument `x`. If x ==y, the result is x. Otherwise, the result is the neighbor of x in the direction of y.
ieee_rem(x,y)	Returns a real with the same type parameter of whichever argument has the greater precision.
ieee_rint(x,y)	Returns a real with the same type parameter as `x`, and whose value is that of x rounded to an integer value according to the current rounding mode.
ieee_scalb(x,i)	Returns a real with the same type parameter as `x`, and whose value is $2^i x$. If $2^i x$ is too large, `ieee_overflow` signals. If $2^i x$ is too small, `ieee_underflow` signals.
ieee_unordered(x,y)	Returns `.true.` if `x` or `y` or both are a NaN.
ieee_value(x,class)	Returns a real with the same type parameter as `x` and a value specified by `class`.

IEEE_ARITHMETIC Non-Elemental Subroutines

The `ieee_arithmetic` intrinsic module supports a number of elemental functions. Table 6.17 provides a list and brief description of what it means if the inquiry function returns .true. In all cases, if the condition for returning .true. is not met, the function returns .false.

In these subroutines, the argument `round_value` is a scalar of type `ieee_round_type` and the argument `gradual` is a scalar of type default `logical`.

Table 6.18. IEEE_ARITHMETIC Non-Elemental Subroutines

This non-elemental subroutine...	Does this...

ieee_get_rounding_mode(round_value)	Returns the floating-point rounding mode.
	If one of the IEEE modes is in operation, the value is one of these:
	`ieee_nearest`
	`ieee_to_zero`
	`ieee_up`
	`ieee_down`
	Otherwise, the value is `ieee_positive_normal`

ieee_get_underflow_mode(gradual)	Returns `.true.` if gradual underflow is in effect-point rounding mode.Otherwise, it returns `.false.`
ieee_set_rounding_mode(round_value)	Specifies the rounding mode to be set.
ieee_set_underflow_mode(gradual)	Sets gradual underflow in effect if the value is `.true.` ; otherwise, gradual underflow ceases to be in effect.

IEEE_ARITHMETIC Transformational Function

The `ieee_arithmetic` intrinsic module supports `ieee_selected_real_kind([p] [,r])` a transformational function that is permitted in an initialization expression.

This result of this function is the kind value of a real `x` for which `ieee_support_datatype(x)` is true.

Module IEEE_EXCEPTIONS

The `ieee_exceptions` intrinsic module provides support for overflow and divide-by-zero flags in the scoping unit for all available kinds of reals and complex data. It also determines the level of support for other exceptions.

This module contains two derived types, named constants of these types, and a collection of generic procedures.

IEEE_EXCEPTIONS Derived Types

- **ieee_flag_type** - Identifies a particular exception flag.
- **ieee_status_type** - Saves the current floating-point status.

For both of these types, the following are true:

- The components are private.
- No operations are defined for these types.
- Only intrinsic assignment is available.

Table 6.19 provides a quick overview of the values that each derived type can take.

Table 6.19. IEEE_EXCEPTIONS Derived Types

This derived type...	Must have one of these values...

ieee_flag_type	For named constants:
	`ieee_overflow`
	`ieee_underflow`
	`ieee_divide_by_zero`
	`ieee_inexact`
	`ieee_invalid`

ieee_status_type	Includes the values of all supported flags as well as current rounding mode.

IEEE_EXCEPTIONS Inquiry Functions

The `ieee_exceptions` intrinsic module supports two inquiry functions, both of which are pure:

- ieee_support_flag(flag [,x])
- ieee_support_halting(flag)

For both functions, the argument flag must be of type `type(ieee_flag_type)`.

Table 6.20 provides a list and brief description of what it means if the inquiry function returns `.true.`I n all cases, if the condition for returning .true. is not met, the function returns `.false.`.

Table 6.20. IEEE_EXCEPTIONS Inquiry Functions

This inquiry function...	Returns .true. if ...
ieee_support_flag(flag [,x])	The processor supports the exception `flag` for all reals. If the optional argument `x` is present, then it returns `.true.` if the processor supports the exception `flag` for all reals of the same kind type parameter as the real argument `x`.
ieee_support_halting(flag)	The processor supports the ability to change the mode by `call ieee_set_halting(flag).`

IEEE_EXCEPTIONS Subroutines Functions

The `ieee_exceptions` intrinsic module supports elemental and non-elemental subroutines.

In all these subroutines:

- `flag` is of type `type(ieee_flag_type)`
- `halting` is of type default logical
- `flag_value` is of type default logical
- `status_value` if is type `type(ieee_status_type)`.

Elemental Subroutines

Table 6.21 provides a list and brief description of what it means if the inquiry function returns `.true.` In all cases, if the condition for returning .true. is not met, the function returns `.false.`.

Table 6.21. IEEE_EXCEPTIONS Elemental Subroutines

This elemental subroutine...	Does this...
ieee_get_flag(flag, flag_value)	If the value of flag is `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_underflow`, or `ieee_inexact` and the corresponding exception flag is signaling, `flag_value` is `true`.
ieee_get_halting_mode(flag, halting)	The value flag must have one of the values: `ieee_invalid`, `ieee_overflow`, `ieee_divide_by_zero`, `ieee_underflow`, or `ieee_inexact`. If the exception specified causes halting, `halting` is `true`.

Non-Elemental Subroutines

The `ieee_exceptions` intrinsic module supports non-elemental subroutines for flags and halting mode as well as for floating-point status.

Table 6.21 provides a list and brief description of these subroutines.

Table 6.22. IEEE_EXCEPTIONS Elemental Subroutines

This non-elemental subroutine...	Does this...
ieee_set_flag(flag, flag_value)	If the value returned by `ieee_support_halting` is true, each `flag` specified is set to be signalling if the corresponding `flag_value` is `true` and is set to be quiet if it is `false`.
ieee_set_halting_mode(flag, halting)	Each exception specified by `flag` causes halting if the corresponding value of `halting` is `true`. If value is false, it does not cause halting.
ieee_get_status(status_value)	Returns the floating-point status, including all the exception flags, the rounding mode, and the halting mode.
ieee_set_status(status_value)	Resets the floating-point status, including all the exception flags, the rounding mode, and the halting mode to the previous invocation of `ieee_get_status`.

IEEE_FEATURES

The `ieee_features` intrinsic module supports specification of essential IEEE features. It provides access to one derived type and a collection of named constants of this type, each of which corresponds to an IEEE feature. The named constants affect the manner in which code is compiled in the scoping units.

IEEE_FEATURES Derived Type

The `ieee_features` intrinsic module provides access to the derived type: `ieee_features_type`. This type identifies a particular feature. It may only take values that are those of named constants defined in the module.

While permitted, there is no purpose in declaring data of type `ieee_features_type`. The components of this type are private, no operation is defined for it, and only intrinsic assignment is available for it.

IEEE_FEATURES Named Constants

Table 6.23 lists a complete set of names constants available for the `ieee_features` intrinsic module and provides the effect of their accessibility:

Table 6.23. IEEE_FEATURES Named Constants

This named constant...	Requires the scoping unit to support ...
ieee_datatype	`ieee_ARITHMETIC` for at least one kind of real.
ieee_denormal	Denormalized numbers for at least one kind of real.
ieee_divide	IEEE divide for at least one kind of real.
ieee_halting	Control of halting for each flag supported.
ieee_inexact_flag	Inexact exception for at least one kind of real.
ieee_inf	Infinity and -infinity for at least one kind of real.
ieee_invalid_flag	Invalid exception for at least one kind of real.
ieee_nan	NaNs for at least one kind of real.
ieee_rounding	Control of the rounding mode for all four rounding modes on at least one kind of real.
ieee_sqrt	IEEE square root for at least one kind of real.
ieee_underflow_flag	Underflow exception for at least one kind of real.

Note

Some features may slow execution on some processorts. Therefore, if `ieee_exceptions` is accessed but `ieee_features` is not, the processor can support a selected subset of the features.

Module iso_c_binding

The `iso_c_binding` intrinsic module provides access to named constants of type default integer that represent kind type parameters of data representations compatible with C types.

- A positive value indicates that the Fortran type and kind type parameter interoperate with the corresponding C type.
- A negative value indicates a lack of support.

Module iso_fortran_env

The `iso_fortran_env` intrinsic module provides information about the Fortran environment through named constants. The following table provides the constants and a brief description of the information provided. Each named constant is a default integer scalar.

Table 6.24. iso_fortran_env Named Constants

This Named Constant...	Provides this Fortran environment information...
character_storage_size	The size, in bits, of a character storage unit
error_unit	The unit number for a preconnected output unit suitable for reporting errors.
file_storage_size	The size, in bits, of a file storage unit.
input_unit	The unit number for the preconnected external unit used for input.
iostat_end	The value returned by IOSTAT= that indicates an end-of-file condition occurs during execution of a READ statement.
iostat_eor	The value returned by IOSTAT= that indicates an end-of-record condition occurs during execution of a READ statement.
numeric_storage_size	The size, in bits, of a numeric storage unit.
output_unit	The unit number for the preconnected external unit used for output.

These special unit numbers may be negative, though they are never -1, since -1 is reserved for another purpose.

The error-unit may be the same as output-unit.

Chapter 7. Object Oriented Programming

Object-oriented programming, OOP, describes an approach to programming where a program is viewed as a collection of interacting, but mostly independent software components. These software components, known as objects, are typically implemented as an entity that encapsulates both data and procedures. Object-oriented programming focuses on the data structures; that is, focus is on the objects on which the program operates rather than the procedures. In languages designed to be object-oriented, there are *classes*, containing both data and modules, that operate on that data. In Fortran, modules may contain data, but there is no notion of separate instances of a module. However, in Fortran 2003, there are type extensions and type-bound procedures that support an object-oriented approach. To have "class-like" behavior, you can combine a module, which contains the methods that operate on the "class", with a derived type containing the data.

PGI Fortran compilers contain procedures, functions, and attributes from Fortran 2003 that facilitate an object-oriented approach to programming. Some of the object-oriented features include classes, type extensions, polymorphic entities, typed allocation, sourced allocation, inheritance association, as well as object-oriented intrinsics. This section provides a high-level overview of these features.

Tip

For specific information on how to use these extensions and for examples, refer to one of the many reports and texts available, such as these:

- *Object-Oriented Programming in Fortran 2003*, PGI Insider, February 2011
- *The Fortran 2003 Handbook: The Complete Syntax, Features and Procedures* by Adams, J.C., Brainerd, W.S., Hendrickson, R.A., Maine, R.E., Martin, J.T., Smith, B.T
- *Fortran 95/2003 explained* by Metcalf,m., Reid, J., and Cohen, M.

Inheritance

Inheritance allows code reusability through an implied inheritance link in which leaf objects, known as children, reuse components from their parent and ancestor objects.

For example, the following code shows how a square type inherits components from rectangle which inherits components from shape.

Example 7.1. Inheritance of Shape Components

```
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
end type shape
type, EXTENDS ( shape ) :: rectangle
        integer :: length
        integer :: width
end type rectangle
type, EXTENDS ( rectangle ) :: square
end type square
```

The programmer indicates the inheritance relationship with the EXTENDS keyword followed by the name of the parent type in parentheses. A type that EXTENDS another type is known as a type extension (e.g., rectangle is a type extension of shape, square is a type extension of rectangle and shape). A type without any EXTENDS keyword is known as a base type (e.g., shape is a base type).

A type extension inherits all of the components of its parent (and ancestor) types. A type extension can also define additional components as well. For example, rectangle has a length and width component in addition to the color, filled, x, and y components that were inherited from shape. The square type, on the other hand, inherits all of the components from rectangle and shape, but does not define any components specific to square objects. The following example show how to access the color component of square:

```
type(square) :: sq               ! declare sq as a square object
   sq%color                      ! access color component for sq
   sq%rectangle%color            ! access color component for sq
   sq%reactangle%shape%color     ! access color component for sq
```

There are three different ways for accessing the color component for sq. A type extension includes an implicit component with the same name and type as its parent type. This approach is handy when the programmer wants to operate on components specific to a parent type. It also helps illustrate an important relationship between the child and parent types.

We often say the child and parent types have a "is a" relationship. In the shape example, "a square *is a* rectangle", "a rectangle is a shape", "a square *is a* shape", and "a shape *is a* base type". This relationship also applies to the type itself: "a shape is a shape", "a rectangle is a rectangle", and "a square is a square").

The "is a" relationship does not imply the converse. A rectangle is a shape, but a shape is not a rectangle since there are components found in rectangle that are not found in shape. Furthermore, a rectangle is not a square because square has a component not found in rectangle; the implicit rectangle parent component.

Polymorphic Entities

Polymorphism permits code reusability in the Object-Oriented Programming paradigm because the programmer can write procedures and data structures that can operate on a variety of data types and values. The programmer does not have to reinvent the wheel for every data type a procedure or a data structure will encounter.

The "is a" relationship might help you visualize how polymorphic variables interact with type extensions. A polymorphic variable is a variable whose data types may vary at run time. Polymorphic entities must be a pointer or allocatable variable or a dummy data object.

There are two basic types of polymorphism:

procedure polymorphism
: Procedure polymorphism deals with procedures that can operate on a variety of data types and values.

data polymorphism
: Data polymorphism deals with program variables that can store and operate on a variety of data types and values. You see later that the dynamic type of these variables changes when we assign a target to a polymorphic pointer variable or when we use typed or sourced allocation with a polymorphic allocatable variable.

To declare a polymorphic variable, use the `class` keyword.

Example 7.2. Polymorphic Variables

In this example, the *sh* object can be a pointer to a shape or any of its type extensions. So, it can be a pointer to a shape, a rectangle, a square, or any future type extension of shape. As long as the type of the pointer target "is a" shape, *sh* can point to it.

```
class(shape), pointer :: sh
```

This second example shows how to declare a pointer `p` that may point to any object whose type is in the class of types or extensions of the type `type(point)`

```
type point
    real :: x,y
end type point
class(point), pointer :: p
```

Unlimited Polymorphic Entities

Unlimited polymorphic entities allow the user to have a pointer that may refer to objects of any type, including non-extensible or intrinsic types. You can use unlimited polymorphic objects to create heterogeneous data structures, such as a list object that links together a variety of data types. Further, you can use abstract types to dictate requirements for type extensions and how they interact with polymorphic variables.

Note

Unlimited polymorphic entities can only be used as an actual argument, as the pointer or target in a pointer assignment, or as the selector in a SELECT TYPE statement.

To declare an unlimited polymorphic variable, use the `*` as the `class` specifier. The following example shows how to declare `up` as an unlimited polymorphic pointer and associate it with a real target.

```
class(*), pointer :: up
  real, target :: x,
  :
  up => x
```

Typed Allocation for Polymorphic Variables

The ALLOCATE statement allows the user to specify the type of polymorphic variables It allocates storage for each allocatable array, pointer object, or pointer-based variable that appears in the statements; declares storage for deferred-shape arrays.

Sourced Allocation for Polymorphic Variables

Sourced allocation defines the type, type parameters, and value of a variable by using the type, type parameters and value of another variable or expression. This type of allocation produces a “clone” of the source expression. To do this, use the ALLOCATE statement, specifying the source of the values through the `source=` clause of that statement.

Procedure Polymorphism

Procedure polymorphism occurs when a procedure, such as a function or a subroutine, can take a variety of data types as arguments. In F2003, this procedure is one that has one or more dummy arguments declared with the CLASS keyword.

In the following example, the `setColor` subroutine takes two arguments, `sh` and `color`. The `sh` dummy argument is polymorphic, based on the usage of `class(shape)`.

```
subroutine setColor(sh, color)
  class(shape) :: sh
  integer :: color
  sh%color = color
  end subroutine setColor
```

The setColor subroutine takes two arguments, `sh` and `color`. The `sh` dummy argument is polymorphic, based on the usage of `class(shape)`.

The subroutine can operate on objects that satisfy the "is a" shape relationship. So, setColor can be called with a shape, rectangle, square, or any future type extension of shape. However, by default, only those components found in the declared type of an object are accessible. For example, shape is the declared type of `sh`. Therefore, you can only access the shape components, by default, for `sh` in setColor (i.e., sh%color, sh %filled, sh%x, sh%y).

If a programmer needs to access the components of the dynamic type of an object, the F2003 SELECT TYPE construct is useful. The following example illustrates how a SELECT TYPE construct can access the components of a dynamic type of an object.

Example 7.3. SELECT TYPE construct

```
subroutine initialize(sh, color, filled, x, y, length, width)
! initialize shape objects
class(shape) :: sh
    integer :: color
    logical :: filled
    integer :: x
```

```
    integer :: y
    integer, optional :: length
    integer, optional :: width

    sh%color = color
    sh%filled = filled
    sh%x = x
    sh%y = y
```

```
select type (sh)
type is (shape)
      ! no further initialization required
class is (rectangle)
    ! rectangle or square specific initializations
    if (present(length))  then
       sh%length = length
    else
       sh%length = 0
    endif
    if (present(width)) then
        sh%width = width
    else
        sh%width = 0
    endif
class default
  ! give error for unexpected/unsupported type
     stop 'initialize: unexpected type for sh object!'
end select
end subroutine initialize
```

The preceding example illustrates an initialization procedure for our shape example. It takes one shape argument, `sh`, and a set of initial values for the components of `sh`. Two optional arguments, `length` and `width`, are specified when we want to initialize a rectangle or a square object.

The SELECT TYPE construct allows us to perform a type check on an object. There are two styles of type checks that we can perform.

- The first type check is called "type is". This type test is satisfied if the dynamic type of the object is the same as the type specified in parentheses following the "type is" keyword.
- The second type check is called "class is". This type test is satisfied if the dynamic type of the object is the same or an extension of the specified type in parentheses following the "class is" keyword.

In the example, if the type of `sh` is rectangle or square. then it initializes the length and width fields. If the dynamic type of `sh` is not a shape, rectangle, or square, then it executes the "class default" branch. This branch may also get executed if the shape type is extended without updating the initialize subroutine.

With the addition of a "class default" branch, the `type is` (shape) branch is needed, even though it does not perform any additional assignments. Otherwise, this example would incorrectly print an error message when `sh` is of type shape.

Procedure Polymorphism with Type-Bound Procedures

Derived types in F2003 are considered objects because they encapsulate data as well as procedures. Procedures encapsulated in a derived type are called *type-bound procedures*. The following example illustrates how to add a type-bound procedure to shape:

```
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
        procedure :: initialize
end type shape
```

F2003 added a `contains` keyword to its derived types to separate a type's data definitions from its procedures. Anything that appears after the `contains` keyword in a derived type must be a type-bound procedure declaration.

Syntax of type-bound procedure declaration:

```
PROCEDURE [(interface-name)] [[,binding-attr-list ]::] binding-name[=> procedure-name]
```

At the minimum, a type-bound procedure is declared with the PROCEDURE keyword followed with a `binding-name`.

The `binding-name` is the name of the type-bound procedure.

The first option is `interface-name`.

The `binding-attr-list` option is a list of binding-attributes.

- PASS and NOPASS attributes allow the procedure to specify to which argument, if any, the invoking object is passed. For example, `pass(x)` passes it to dummy argument x, while `nopass` indicates not to pass it at all.
- NON_OVERRIDABLE attribute specifies that the type-bound procedure cannot be overridden during type extension.
- PRIVATE and PUBLIC attributes determine where the type-bound procedures can be referenced. The default is `public`, which allows the procedures to be referenced anywhere in the program having that type of variable. If the procedure is `private`, it can only be referenced from within the module in which it is defined.
- DEFERRED are type bound procedures that are declared in an abstract type. as described in "Abstract Types and Deferred Bindings," on page 145, and must be defined in all of its non-abstract type extensions.

The `procedure-name` option is the name of the underlying procedure that implements the type-bound procedure. This option is required if the name of the underlying procedure differs from the binding-name. The procedure-name can be either a module procedure or an external procedure with an explicit interface.

In Example 7.3, "SELECT TYPE construct," on page 126, the binding-name is `initialize`. Because `procedure-name` was not specified, an implicit `procedure-name`, `initialize`, is also declared. Another way to write that example is `procedure :: initialize => initialize`.

Example 7.4. Type-Bound Procedure using Module Procedure

The following example is a type-bound procedure that uses a module procedure.

```
module shape_mod
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
        procedure :: initialize
end type shape
type, extends(shape) :: rectangle
        integer :: length
        integer :: width
end type rectangle
type, extends(rectangle) :: square
end type square
contains
subroutine initialize(sh, color, filled, x, y, length, width)
! initialize shape objects
class(shape) :: sh
integer :: color
logical :: filled
integer :: x
integer :: y
integer, optional :: length
integer, optional :: width

sh%color = color
sh%filled = filled
sh%x = x
sh%y = y
select type (sh)
type is (shape)
      ! no further initialization required
class is (rectangle)
    ! rectangle or square specific initializations
    if (present(length))  then
       sh%length = length
    else
       sh%length = 0
    endif
    if (present(width)) then
        sh%width = width
    else
        sh%width = 0
    endif
class default
  ! give error for unexpected/unsupported type
     stop 'initialize: unexpected type for sh object!'
end select
end subroutine initialize
end module
```

Example 7.5. Type-Bound Procedure using an External Procedure

The following example is a type-bound procedure that uses an external procedure with an explicit interface:

```
module shape_mod
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
        procedure :: initialize
end type shape
type, extends(shape) :: rectangle
        integer :: length
        integer :: width
end type rectangle
type, extends(rectangle) :: square
end type square
interface
    subroutine initialize(sh, color, filled, x, y, length, width)
    import shape
    class(shape) :: sh
    integer :: color
    logical :: filled
    integer :: x
    integer :: y
    integer, optional :: length
    integer, optional :: width
  end subroutine
end interface
end module
```

Using the preceding examples, we can invoke the type-bound procedure in the following manner:

```
use shape_mod
type(shape) :: shp                      ! declare an instance of shape
call shp%initialize(1, .true., 10, 20)   ! initialize shape
```

The syntax for invoking a type-bound procedure is very similar to accessing a data component in a derived type. The name of the component is preceded by the variable name separated by a percent (%) sign. In this case, the name of the component is `initialize` and the name of the variable is `shp`. To access the initialize type-bound procedure, type `shp%initialize`. Using the preceding invocation calls the `initialize` subroutine and passes in `1` for color, `.true.` for filled, `10` for x, and `20` for y.

But what about the first dummy argument, `sh`, in initialize? This dummy argument is known as the *passed-object dummy* argument. By default, the passed-object dummy is the first dummy argument in the type-bound procedure. It receives the object that invoked the type-bound procedure. In our example, `sh` is the passed-object dummy and the invoking object is `shp`. Therefore, the `shp` object gets assigned to `sh` when `initialize` is invoked.

The passed-object dummy argument must be declared CLASS and of the same type as the derived type that defined the type-bound procedure. For example, a type bound procedure declared in shape must have a passed-object dummy argument declared "class(shape)".

You can also specify a different passed-object dummy argument using the PASS binding-attribute. For example, suppose that the `sh` dummy in our initialize subroutine did not appear as the first argument. Then you must specify a PASS attribute, as illustrated in the following code:

```
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
        procedure, pass(sh) :: initialize
end type shape
```

If you do not want to specify a passed-object dummy argument, you can do so using the NOPASS binding-attribute:

```
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
        procedure, nopass :: initialize
end type shape
```

When you specify NOPASS, you invoke the type-bound procedure the same way. The only difference is that the invoking object is not automatically assigned to a passed-object dummy in the type-bound procedure. For example, if you were to specify NOPASS in the `initialize` type-bound procedure, then you would invoke it this way:

```
type(shape) :: shp                          ! declare an instance of shape
call shp%initialize(shp, 1, .true., 10, 20) ! initialize shape
```

You explicitly specify `shp` for the first argument of initialize because it was declared NOPASS.

Inheritance and Type-Bound Procedures

A child type inherits or reuses components from their parent or ancestor types. When dealing with F2003 derived types, this inheritance applies to both data and procedures. In the following example, rectangle and square both inherit the `initialize` type-bound procedure from `shape`.

```
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
        procedure :: initialize
end type shape
type, EXTENDS ( shape ) :: rectangle
        integer :: length
        integer :: width
end type rectangle
type, EXTENDS ( rectangle ) :: square
end type square
```

Using the example above, we can invoke initialize with a shape, rectangle, or square object:

```
type(shape) :: shp                                  ! declare an instance of shape
type(rectangle) :: rect                             ! declare an instance of rectangle
type(square) :: sq                                  ! declare an instance of square
call shp%initialize(1, .true., 10, 20)              ! initialize shape
call rect%initialize(2, .false., 100, 200, 50, 25) ! initialize rectangle
call sq%initialize(3, .false., 400, 500, 30, 20)   ! initialize rectangle
```

Procedure Overriding

Most OOP languages allow a child object to override a procedure inherited from its parent object. This is known as procedure overriding. In F2003, you can specify a type-bound procedure in a child type that has the same binding-name as a type-bound procedure in the parent type. When the child overrides a particular type-bound procedure, the version defined in its derived type is invoked instead of the version defined in the parent. In the following example, `rectangle` defines an `initialize` type-bound procedure that overrides shape's `initialize` type-bound procedure.

```
module shape_mod
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
        procedure :: initialize => initShape
end type shape
type, EXTENDS ( shape ) :: rectangle
        integer :: length
        integer :: width
   contains
        procedure :: initialize => initRectangle
end type rectangle
type, EXTENDS ( rectangle ) :: square
end type square
contains
subroutine initShape(this, color, filled, x, y, length, width)
! initialize shape objects
class(shape) :: this
integer :: color
logical :: filled
integer :: x
integer :: y
integer, optional :: length  ! ingnored for shape
integer, optional :: width   ! ignored for shape
this%color = color
this%filled = filled
this%x = x
this%y = y
end subroutine

subroutine initRectangle(this, color, filled, x, y, length, width)
! initialize rectangle objects
class(rectangle) :: this
integer :: color
logical :: filled
integer :: x
integer :: y
integer, optional :: length
integer, optional :: width
this%color = color
```

```
this%filled = filled
this%x = x
this%y = y
if (present(length)) then
   this%length = length
else
   this%length = 0
endif
if (present(width)) then
    this%width = width
else
     this%width = 0
endif
end subroutine
end module
```

The preceding example illustrates code that defines a type-bound procedure called `initialize` for both shape and rectangle. The only difference is that shape's version of initialize invokes a procedure called `initShape` while rectangle's version invokes a procedure called `initRectangle`. The passed-object `dummy` in `initShape` is declared "class(shape)" and the passed-object dummy in `initRectangle` is declared "class(rectangle)".

A type-bound procedure's passed-object `dummy` must match the type of the derived type that defined it. Other than differing passed-object `dummy` arguments, the interface for the child's overriding type-bound procedure is identical with the interface for the parent's type-bound procedure. Both type-bound procedures are invoked in the same manner:

```
type(shape) :: shp                          ! declare an instance of shape
type(rectangle) :: rect                      ! declare an instance of rectangle
type(square) :: sq                          ! declare an instance of square
call shp%initialize(1, .true., 10, 20)              ! calls initShape
call rect%initialize(2, .false., 100, 200, 11, 22)  ! calls initRectangle
call sq%initialize(3, .false., 400, 500)            ! calls initRectangle
```

`sq` is declared square but its initialize type-bound procedure invokes `initRectangle` because `sq` inherits the rectangle version of initialize.

Although a type may override a type-bound procedure, it is still possible to invoke the version defined by a parent type. Each type extension contains an implicit parent object of the same name and type as the parent. You can use this implicit parent object to access components specific to a parent, say, a parent's version of a type-bound procedure, as illustrated here:

```
call rect%shape%initialize(2, .false., 100, 200)         ! calls initShape
call sq%rectangle%shape%initialize(3, .false., 400, 500) ! calls initShape
```

If you do not want a child to override a parent's type-bound procedure, you can use the NON_OVERRIDABLE binding-attribute to prevent any type extensions from overriding a particular type-bound procedure:

```
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
        procedure, non_overridable :: initialize
end type shape
```

Functions as Type-Bound Procedures

In the preceding examples, subroutines implement type-bound procedures. You can also implement type-bound procedures with functions. The following example uses a function that queries the status of the filled component in shape.

```
module shape_mod
type shape
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
        procedure :: isFilled
end type shape
contains
     logical function isFilled(this)
     class(shape) :: this
     isFilled = this%filled
     end function
end module
```

You can invoke the preceding function in the following manner:

```
use shape_mod
type(shape) :: shp        ! declare an instance of shape
logical filled
call shp%initialize(1, .true., 10, 20)
filled = shp%isFilled()
```

Information Hiding

In "Procedure Overriding," on page 132, you saw how a child type can override a parent's type-bound procedure. This process allows a user to invoke a type-bound procedure without any knowledge of the implementation details of that procedure. This is another important feature of Object Oriented Programming know as *information hiding*.

Information hiding allows the programmer to view an object and its procedures as a "black box". That is, the programmer can use (or reuse) an object without any knowledge of the implementation details of the object.

Inquiry functions, like the `isFilled` function, shown in "Functions as Type-Bound Procedures," on page 134, are common with information hiding. The motivation for inquiry functions, rather than direct access to the underlying data, is that the object's implementer can change the underlying data without affecting the programs that use the object. Otherwise, each program that uses the object would need to be updated whenever the underlying data of the object changes.

To enable information hiding, F2003 provides a PRIVATE keyword and binding-attribute. To enable information hiding, F2003 also provides a PUBLIC keyword and binding-attribute. By default, all derived type components are declared PUBLIC. The PRIVATE keyword can be placed on derived type data and type-bound procedure components and on module data and procedures. The following sample illustrates use of PUBLIC and PRIVATE:

Example 7.6. Code Using Private and Public

```
module  shape_mod
private    ! hide the type-bound procedure implementation procedures
public :: shape, constructor   ! allow access to shape & constructor procedure
type shape
   private               ! hide the underlying details
      integer :: color
      logical :: filled
      integer :: x
      integer :: y
   contains
   private                 ! hide the type bound procedures by default
   procedure :: initShape  ! private type-bound procedure
      procedure, public :: isFilled ! allow access to isFilled type-bound procedure
      procedure, public :: print ! allow access to print type-bound procedure
end type shape
contains
logical function isFilled(this)
class(shape) :: this

isFilled = this%filled

end function

function constructor(color, filled, x, y)
type(shape) :: constructor
integer :: color
logical :: filled
integer :: x
integer :: y
 call constructor%initShape(color, filled, x, y)
end function
subroutine initShape(this, color, filled, x, y)
! initialize shape objects
class(shape) :: this
integer :: color
logical :: filled
integer :: x
integer :: y

this%color = color
this%filled = filled
this%x = x
this%y = y
end subroutine

subroutine print(this)
class(shape) :: this
print *, this%color, this%filled, this%x, this%y

end subroutine
end module
```

The preceding example uses information hiding in the host module as well as in the shape type. The *private* statement, located at the top of the module, enables information hiding on all module data and procedures. The `isFilled` module procedure, which is not to be confused with the `isFilled` type-bound procedure,

is hidden as a result of the private statement at the top of the module. The `public :: constructor` allows the user to invoke the constructor module procedure. There is also a private statement on the data components of shape. Now, the only way a user can query the filled component is through the `isFilled` type-bound procedure, which is declared *public*.

Notice the *private* statement after the contains in type shape. The *private* that appears after type shape only affects the data components of shape. If you want your type-bound procedures to also be private, then a *private* statement must also be added after the `contains` keyword. Otherwise, type-bound procedures are public by default.

In Example 7.6, "Code Using Private and Public," on page 135, the `initShape` type-bound procedure is declared *private*. Therefore, only procedures local to the host module can invoke a private type-bound procedure. The constructor module procedure invokes the `initShape` type-bound procedure. You may invoke this example in this way:

```
program shape_prg
use shape_mod
type(shape) :: sh
logical filled
sh = constructor(5, .true., 100, 200)
call sh%print()
end
```

Here is a sample compile and sample run of the preceding program. In this example, the `shape_mod` module is saved in a file called `shape.f03` and the main program is called `main.f03`:

```
% pgfortran -V ; pgfortran shape.f03 main.f03 -o shapeTest
pgfortran 11.2-1 64-bit target on x86-64 Linux -tp penryn
Copyright 2013 NVIDIA Corporation All Rights Reserved.
shape.f03:
main.f03:
% shapeTest
            5  T          100          200
```

Type Overloading

Example 7.6, "Code Using Private and Public," on page 135 creates an instance of shape by invoking a function called `constructor`. This function hides the details for constructing a shape object, including the underlying type-bound procedure that performs the initialization. However, you may have noticed that the word *constructor* could very well be defined somewhere else in the host program. If that is the case, the program cannot use our module without renaming one of the constructor functions. Since OOP encourages information hiding and code reusability, it would make more sense to come up with a name that probably is not being defined in the host program. That name is the type name of the object we are constructing.

F2003 allows the programmer to overload a name of a derived type with a generic interface. The generic interface acts as a wrapper for our constructor function. The idea is that the user would then construct a shape in the following manner:

```
program shape_prg
use shape_mod
type(shape) :: sh
logical filled
sh = shape(5, .true., 100, 200)  ! invoke constructor through shape generic interface
```

```
call sh%print()
end
```

Here is the modified version of Example 7.6, "Code Using Private and Public," on page 135 that uses type overloading:

```
module shape_mod
private    ! hide the type-bound procedure implementation procedures
public :: shape ! allow access to shape
type shape
   private               ! hide the underlying details
        integer :: color
        logical :: filled
        integer :: x
        integer :: y
   contains
   private                 ! hide the type bound procedures by default
   procedure :: initShape  ! private type-bound procedure
   procedure, public :: isFilled ! allow access to isFilled type-bound procedure
end type shape
interface shape
procedure constructor       ! add constructor to shape generic interface
end interface
contains
     :
     :
end module
```

The `constructor` function is now declared private and is invoked through the shape public generic interface.

Data Polymorphism

As described in "Polymorphic Entities," on page 124, the `class` keyword allows F2003 programmers to create a polymorphic variable, that is, a variable whose data type is dynamic at runtime. Recall that the polymorphic variable must be a pointer variable, allocatable variable, or a dummy argument.

Pointer Polymorphic Variables

The following example illustrates pointer polymorphic variables.

```
subroutine init(sh)
class(shape) :: sh              ! polymorphic dummy argument
class(shape), pointer :: p      ! polymorphic pointer variable
class(shape), allocatable:: als ! polymorphic allocatable variable
end subroutine
```

In the preceding example, the `sh`, `p`, and `als` polymorphic variables can each hold values of type `shape` or any type extension of shape.

- The `sh` dummy argument receives its type and value from the actual argument to `sh` of subroutine `init()`. In the same manner that polymorphic dummy arguments form the basis to procedure polymorphism, polymorphic pointer and allocatable variables form the basis to data polymorphism.
- The polymorphic pointer variable `p` can point to an object of type shape or any of its extensions. For example, the `select type` construct in the following example helps illustrate the fact that the

polymorphic pointer, p, can take on several types. In this case, p can point to a shape, rectangle, or square object. The dynamic type of pointer p is not known until the pointer assignment, p => sh in this example, is executed.

```
subroutine init(sh)
class(shape),target :: sh
class(shape), pointer :: p
select type (sh)
type is (shape)
  p => sh
  :  ! shape specific code here
type is (rectangle)
  p => sh
  :  ! rectangle specific code here
type is (square)
  p => sh
  :  ! square specific code here
class default
  p => null()
end select
:
end subroutine
```

Allocatable Polymorphic Variables

The following example illustrates pointer polymorphic variables.

An allocatable polymorphic variable receives its type and optionally its value at the point of its allocation. By default, the dynamic type of a polymorphic allocatable variable is the same as its declared type after executing an allocate statement.

The following example allocates the polymorphic variable als. This variable receives dynamic type shape after the ALLOCATE statement is executed.

```
class(shape), allocatable :: als
allocate(als)
```

Obviously there is not much use for polymorphic allocatable variables if you can only specify the declared type in an allocate statement. Therefore, F2003 provides typed allocation to allow the programmer to specify a type other than the declared type in an allocate statement.

In the following allocate statement, notice that following the type is a :: and then the variable name.

```
class(shape), allocatable :: als
allocate(rectangle::als)
```

In this example, rectangle is the dynamic type of variable als. However, the declared type of als is still shape.

The type specification must be the same or a type extension of the declared type of the allocatable variable. The following example illustrates how to allocate a polymorphic variable with the same type of another object:

```
subroutine init(sh)
class(shape) :: sh
class(shape), allocatable :: als
```

```
select type (sh)
type is (shape)
  allocate(shape::als)
type is (rectangle)
  allocate(rectangle::als)
type is (square)
  allocate(square::als)
end select
:
end subroutine
```

You can expand the preceding example to create a "copy" of an object, as shown here:

```
subroutine init(sh)
class(shape) :: sh
class(shape), allocatable :: als
select type (sh)
type is (shape)
  allocate(shape::als)
  select type(a)
  type is (shape)
    als = sh   ! copy sh
  end select
type is (rectangle)
  allocate(rectangle::als)
   select type (a)
   type is (rectangle)
     als = sh ! copy sh
   end select
type is (square)
  allocate(square::als)
  select type (als)
  type is (square)
    als = sh   ! copy sh
  end select
end select
:
end subroutine
```

The programmer can only access the components of the declared type by default. Therefore, in the preceding example, you can only access the shape components for object `als` by default. To access the components of the dynamic type of object `als` requires you to use a nested select type for object `als`.

The previous example illustrates one application of data polymorphism: making a copy or a clone of an object. Unfortunately, this approach does not scale well if `shape` has several type extensions. Further, whenever a type extension to shape is added, the programmer must update the `init()` subroutine to include the new type extension. An alternative to this is sourced allocation.

Sourced Allocation

Sourced allocation allows you to make an extra copy, or clone, of an object. In the following example, the ALLOCATE statement allocates `als` with the same dynamic type as `sh` and with the same value(s) of `sh`. The `source=` argument specifies the object that you wish to clone.

The declared type of the `source=` must be the same or a type extension of the allocate argument (e.g., `als`).

```
subroutine init(sh)
class(shape) :: sh
class(shape), allocatable :: als
allocate(als, source=sh) ! als becomes a clone of sh
:
end subroutine
```

Unlimited Polymorphic Objects

Data polymorphism using derived types and their type extensions satisfies most applications. However, sometimes you may want to write a procedure or a data structure that can operate on any type, including any intrinsic or derived type. As described in the section on procedure polymorphism, F2003 provides unlimited polymorphic objects.

Here are some examples of unlimited polymorphic objects:

```
subroutine init(sh)
class(*) :: sh              ! unlimited polymorphic dummy argument
class(*), pointer :: p      ! unlimited polymorphic pointer variable
class(*), allocatable:: als ! unlimited polymorphic allocatable variable
end subroutine
```

You use the `class(*)` keyword to specify an unlimited polymorphic object declaration. The operations for unlimited polymorphic objects are similar to those in the preceding section for "limited" polymorphic objects. However, unlike "limited" polymorphic objects, their "unlimited" counterparts can take any F2003 type.

The following example illustrates unlimited polymorphic objects that can be used with procedure polymorphism:

```
subroutine init(sh)
class(*) :: sh
select type(sh)
class is (shape)
  :   ! shape specific code
type is (integer)
  : ! integer specific code
type is (real)
  : ! real specific code
type is (complex)
  : ! complex specific code
end select
end subroutine
```

Similarly, you can assign any pointer or target to an unlimited polymorphic pointer, regardless of type.

The following example shows `sh` assigned to pointer `p`. Then a `select type` construct is used to query the dynamic type of pointer `p`.

```
subroutine init(sh)
class(*),target :: sh
class(*), pointer :: p
p => sh
select type(p)
class is (shape)
  :   ! shape specific code
type is (integer)
  : ! integer specific code
```

```
type is (real)
  : ! real specific code
type is (complex)
  : ! complex specific code
end select
end subroutine
```

You can also use unlimited polymorphic objects with typed allocation. In fact, a type (or `source=`) argument must be specified with the ALLOCATE statement since there is no default type for `class(*)`. However, unlike their "limited" counterparts, as illustrates in the following example, you can specify any F2003 type, intrinsic or derived.

```
subroutine init(sh)
class(*) :: sh
class(*), allocatable :: als
select type(sh)
type is (shape)
  allocate(shape::als)
type is (integer)
  allocate(integer::als)
type is (real)
  allocate(real::als)
type is (complex)
  allocate(complex::als)
end select
:
end subroutine
```

Sourced allocation can also operate on unlimited polymorphic objects:

```
subroutine init(sh)
class(*) :: sh
class(*), allocatable :: als
allocate(als, source=sh) ! als becomes a clone of sh
:
end subroutine
```

If the `source=` argument is an unlimited polymorphic object (i.e., declared class(*)), the allocate argument, in this example `als`, must also be an unlimited polymorphic object.

When the ALLOCATE argument is declared `class(*)`, the declared type in the `source=` argument can be any type including `class(*)`, any derived type, or any intrinsic type.

The following code demonstrates sourced allocation with an unlimited polymorphic allocatable argument and an intrinsic typed `source=` argument.

```
class(*), allocatable :: als
integer i
i = 1234
source(als, source=i)
```

Example 7.7. Data Polymorphic Linked List

One of the advantages to unlimited polymorphic objects is that you can create data structures that operate on all data types, both intrinsic and derived in F2003. Traditionally, data stored in a linked list all have the same data type. However, with unlimited polymorphic objects, we can easily create a list that contains a variety of data types and values.

This example creates data structures that can be used to create a heterogeneous list of objects.

1. Start by creating a derived type that will represent each link in our linked list.

```
type link
     class(*), pointer :: value => null()
     type(link), pointer :: next => null()
end type link
```

   This basic link derived type contains an unlimited polymorphic pointer that points to the value of the link followed by a pointer to the next link in the list.

2. Place this derived type into its own module, add a constructor, and add some type-bound procedures to access the value(s).

   Recall that information hiding allows others to use an object without understanding its implementation details.

```
module link_mod
  private    ! information hiding
  public :: link
  type link
     private   ! information hiding
     class(*), pointer :: value => null()
     type(link), pointer :: next => null()
     contains
     procedure :: getValue    ! get value in this link
     procedure :: nextLink    ! get the link after this link
     procedure :: setNextLink ! set the link after this link
  end type link

  interface link
   procedure constructor
  end interface

contains
   function nextLink(this)
    class(link) :: this
    class(link), pointer :: nextLink
      nextLink => this%next
   end function nextLink
    subroutine setNextLink(this,next)
     class(link) :: this
     class(link), pointer :: next
       this%next => next
    end subroutine setNextLink
  function getValue(this)
    class(link) :: this
    class(*), pointer :: getValue
  getValue => this%value
  end function getValue
  function constructor(value, next)
    class(link),pointer :: constructor
    class(*) :: value
    class(link), pointer :: next
    allocate(constructor)
    constructor%next => next
    allocate(constructor%value, source=value)
  end function constructor
end module link_mod
```

This code uses the PRIVATE keywords. Therefore the user of the object must use the `getValue()` function to access the values of each link in our list, the `nextLink()` procedure to access the next link in the list, and `setNextLink()` to add a link after a link. The `getValue()` function returns a pointer to a `class(*)`, meaning it can return an object of any type.

We employ type overloading for the constructor function. Recall that type overloading allows you to create a generic interface with the same name as a derived type. Therefore you can create a constructor function and hide it behind the name of the type.

3. Construct a link in the following manner:

```
class(link),pointer :: linkList
integer v
linkList => link(v, linkList%next)
```

Although you could easily create a linked list with just the preceding link object, the real power of Object Oriented Programming lies in its ability to create flexible and reusable components. However, the user must understand how the list is constructed with the link object; in this example, the link constructor assigns its result to the `linkList` pointer.

4. To take advantage of OOP, create another object called `list` that acts as the "Application Program Interface" or API to the linked list data structure.

```
type list
   class(link),pointer :: firstLink => null() ! first link in list
   class(link),pointer :: lastLink => null()  ! last link in list
   contains
     procedure :: addInteger ! add integer to list
     procedure :: addChar    ! add character to list
     procedure :: addReal    ! add real to list
     procedure :: addValue   ! add class(*) to list
     generic :: add => addInteger, addChar, addReal, addValue
  end type list
```

The list derived type has two data components, `firstlink`, which points to the first link in its list and `lastLink` which points to the last link in the list. The `lastLink` pointer allows the user to easily add values to the end of the list.

There are four type-bound procedures called `addInteger()`, `addChar()`, `addReal()`, and `addValue()`. You use the first three procedures to add an integer, a character, and a real to the linked list respectively. The `addValue()` procedure adds `class(*)` values to the list and is the main add routine. The `addInteger()`, `addChar()`, and `addReal()` procedures are actually just wrappers to the `addValue()` procedure.

The `addInteger()` procedure takes an integer value and allocates a class(*) with that value using sourced allocation.

```
subroutine addInteger(this, value)
    class(list) :: this
    integer value
    class(*), allocatable :: v
    allocate(v,source=value)
    call this%addValue(v)
  end subroutine addInteger
```

The only difference between `addInteger()`, `addChar()`, and `addReal()` is the data type dummy argument, value.

The value from the procedure is passed to the `addValue()` procedure:

```
subroutine addValue(this, value)
    class(list) :: this
    class(*), value
    class(link), pointer :: newLink
    if (.not. associated(this%firstLink)) then
       this%firstLink => link(value, this%firstLink)
       this%lastLink => this%firstLink
    else
       newLink => link(value, this%lastLink%nextLink())
       call this%lastLink%setNextLink(newLink)
       this%lastLink => newLink
    end if
  end subroutine addValue
```

The `addValue()` procedure takes two arguments; a list and a class(*). If the list's `firstlink` is not associated (i.e., points to null()), then add the value to the start of the list by assigning it to the list's `firstlink` pointer. Otherwise, add it after the list's `lastlink` pointer.

Returning to the list type definition, notice the following statement:

```
generic :: add => addInteger, addChar, addReal, addValue
```

This statement uses an F2003 feature known as a *generic-type bound procedure*. These procedures act very much like generic interfaces, except they are specified in the derived-type and only type-bound procedures are permitted in the generic-set. You define a type-bound procedure to be generic by defining a generic statement within the type-bound procedure part. The statement is of the form:

```
generic [[ , access-spec ] ::] generic-spec => tbp-name-list
```

where *tbp-name-list* is a list of the specific type-bound procedures to be included in the generic set. You can use these statements for named generics as well as for operators and assignments.

In the preceding example, you can invoke the add type-bound procedure and either the `addInteger()`, `addChar()`, `addReal()`, or `addValue()` implementations get called. The compiler determines which procedure to invoke based on the data type of the actual arguments. If you pass an integer to the value argument of `add()`, `addInteger()` is invoked, a character value invokes `addChar()`, a real value invokes `addReal()`, and a class(*) value invokes `addValue()`

Here is a simple program that adds values to a list and prints out the values. You can download the complete `list_mod` and `link_mod` modules, which encapsulate the list and link objects respectively.

```
program main
  use list_mod
  implicit none
  integer i
  type(list) :: my_list

  do i=1, 10
     call my_list%add(i)
  enddo
  call my_list%add(1.23)
  call my_list%add('A')
  call my_list%add('B')
```

```
  call my_list%add('C')
  call my_list%printvalues()
end program main
```

```
% pgfortran -c list.f90
% pgfortran -c link.f90
% pgfortran -V main.f90 list.o link.o
pgfortran 11.6-0 64-bit target on x86-64 Linux -tp penryn
Copyright 2013 NVIDIA Corporation All Rights Reserved.
```

```
% a.out
            1
            2
            3
            4
            5
            6
            7
            8
            9
           10
    1.230000
 A
 B
 C
```

Abstract Types and Deferred Bindings

Example 7.7, "Data Polymorphic Linked List" contained a list derived type that acted as the API for a linked list. Rather than employ one implementation for the list derived type, you could choose to define some of the components and type-bound procedures for a list object and require the user to define the rest. One way to do this is through an abstract type.

An *abstract type* is a derived type that cannot be instantiated. Instead, it is extended and further defined by another type. The type extension can also be declared abstract, but ultimately it must be extended by a non-abstract type if it ever is to be instantiated in a program.

The following example illustrates a list type declared abstract:

```
module abstract_list_mod
:
type, abstract :: list
     private
     class(link),pointer :: firstLink => null() ! first link in list
     class(link),pointer :: lastLink => null()  ! last link in list
     class(link),pointer :: currLink => null()  ! list iterator
   contains
     procedure, non_overridable :: addValue    ! add value to list
     procedure, non_overridable :: firstValue  ! get first value in list
     procedure, non_overridable :: reset       ! reset list iterator
     procedure, non_overridable :: next        !iterate to next value in list
     procedure, non_overridable :: currentValue! get current value in list
     procedure, non_overridable :: moreValues  ! more values to iterate?
     generic :: add => addValue
     procedure(printValues), deferred :: printList  ! print contents of list
  end type list
  abstract interface
  subroutine printValues(this)
    import list
```

```
    class(list) :: this
   end subroutine
   end interface
  :
 end module abstract_list_mod
```

The abstract list type in the preceding code uses the link type from Example 7.7, "Data Polymorphic Linked List" as its underlying data structure. This example has three data components, `firstLink`, `lastLink`, and `currLink`.

- The `firstLink` component points to the first link in the list.
- The `lastLink` component points to the last link in the list.
- The `currLink` component points to the "current" link that we are processing in the list. In other words, `currLink` acts as a list iterator that allows us to traverse the list using inquiry functions. Without a list iterator, the user of this list type would need to understand the underlying link data structure. Instead, the code takes advantage of information hiding by providing a list iterator.

Our list type is declared `abstract`. Therefore, the following declaration and allocate statements are invalid for `list`:

```
type(list) :: my_list    ! invalid because list is abstract
allocate(list::x)        ! invalid because list is abstract
```

On the other hand, you can use the abstract type in a class declaration since its dynamic type can be a non-abstract type extension. In the following example, the usage of list is valid because nothing is declared or allocated with type list. Instead, each variable is some type extension of list.

```
subroutine list_stuff(my_list)
class(list) :: my_list
class(list), pointer :: p
class(list), allocatable :: a
select type (my_list)
class is (list)
:
end select
end subroutine
```

The preceding list type definition has the deferred binding added to the `printValues` type-bound procedure. Deferred bindings allow the author of the abstract type to dictate what procedures must be implemented by the user of the abstract type and what may or may not be overridden. You can add the deferred binding to type-bound procedures that are not defined in the abstract type, but these must be defined in all of its non-abstract type extensions. F2003 also requires that a deferred binding have an interface (or an abstract interface) associated with it.

You use the following syntax for deferred bindings:

procedure (interface-name), deferred :: procedure-name

Because deferred bindings have an interface associated with them, there is no => followed by an implementation-name allowed in the syntax. For example, `procedure, deferred :: foo => bar` is not allowed.

The following module includes an `integerList` which extends the abstract type, `list`, previously defined.

```
module integer_list_mod
:
  type, extends(list) :: integerList
   contains
     procedure :: addInteger
     procedure :: printList => printIntegerList
     generic :: add => addInteger
  end type integerList
:
end module integer_list_mod
```

In this example, `printList()` is defined as required by the deferred binding in `list`. You can use the following implementation for the `printList()` type-bound procedure:

```
  subroutine printIntegerList(this)
    class(integerList) :: this
    class(*), pointer :: curr
    call this%reset()              ! reset list iterator
    do while(this%moreValues())    ! loop while there are values to print
       curr => this%currentValue() ! get current value
       select type(curr)
       type is (integer)
          print *, curr            ! print the integer
       end select       call this%nextValue()       ! increment the list iterator
    end do
    call this%reset()              ! reset list iterator
  end subroutine printIntegerList
```

`printIntegerList()` prints the integers in the list. The list reset() procedure verifies that the list iterator is at the beginning of the list. Then the subroutine loops through the list, calling the list's `moreValues()` function to determine if our list iterator has reached the end of the list. The list's `currentValue()` function gets the value to which the list iterator is pointing. A select type accesses the integer value and prints it. Finally, the list's `nextValue()` procedure increments the list iterator to access the next value.

The following a sample program uses the abstract list and `integerList` types. The sample program adds the integers one through ten to the list and then calls the integerList's `printList()` procedure. Next, the program traverses the list, places the integers into an array, and then prints out the array. You can download the complete abstract_list_mod and integer_list_mod modules from the PGI website.

```
program main
  use integer_list_mod
  implicit none
  integer i
  type(integerList) :: my_list
  integer values(10)
  do i=1, 10
     call my_list%add(i)
  enddo
  call my_list%printList()
  print *
  call my_list%reset()
  i = 1
  do while(my_list%moreValues())
     values(i) = my_list%current()
     call my_list%next()
     i = i + 1
  end do
  print *, values
end program main
```

Here is a sample compile and run of the preceding program:

```
% pgfortran -c link.f90
% pgfortran -c abstract_list.f90
% pgfortran -c integerList.f90
% pgfortran -V main.f90 link.o abstract_list.o integerList.o
pgfortran 11.6-0 64-bit target on x86-64 Linux -tp penryn
Copyright 2013 NVIDIA Corporation All Rights Reserved.
```

```
% a.out
           1
           2
           3
           4
           5
           6
           7
           8
           9
          10
           1           2           3           4           5
6          7           8           9          10
```

IEEE Modules

PGI 2013 supports the Fortran IEEE standard intrinsic modules `ieee_arithmetic`, `ieee_exceptions`, and `ieee_features`.

- `ieee_arithmetic` affects the manner in which code is compiled in the scoping units.
- `ieee_exceptions` specifies accessibility of overflow and divide-by-zero flags as well as determines the level of support for other exceptions.
- `ieee_features` supports specification of essential IEEE features. It provides access to one derived type and a collection of named constants of this type that affect the manner in which code is compiled in the scoping units.

For details on each of these modules, refer to "Intrinsic Modules," on page 112.

Intrinsic Functions

Table 7.1 lists the Fortran 2003 intrinsic functions available to facilitate an object-oriented approach to programming. A more complete description of each of these intrinsics is available in Chapter 6, "*Fortran Intrinsics*," on page 89.

Table 7.1. Fortran 2003 Functions and Procedures

Generic Name	Purpose	Num. Args	Argument Type	Result Type
EXTENDS_TYPE_OF	Determines whether the dynamic type of A is an extension type of the dynamic type of B.	2	Objects of extensible type	LOGICAL SCALAR

Generic Name	Purpose	Num. Args	Argument Type	Result Type
MOVE_ALLOC	Moves an allocation from one allocatable object to another.	2	Any - of same type and rank	none
SAME_TYPE_AS	Determines whether the dynamic type of A is the same as the dynamic type of B.	2	Objects of extensible type	LOGICAL SCALAR

Chapter 8. OpenMP Directives for Fortran

The PGF77 and PGFORTRAN compilers support the OpenMP Fortran Application Program Interface. The OpenMP shared-memory parallel programming model is defined by a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran programs.

The directives include a parallel region construct for writing coarse grain SPMD programs, work-sharing constructs which specify that DO loop iterations should be split among the available threads of execution, and synchronization constructs. The data environment is controlled using clauses on the directives or with additional directives. Runtime library routines are provided to query the parallel runtime environment, for example to determine how many threads are participating in execution of a parallel region. Finally, environment variables are provided to control the execution behavior of parallel programs. For more information on OpenMP, refer to this website:

http://www.openmp.org

For an introduction to how to execute programs that use multiple processors along with some pointers to example code, refer to "Parallel Programming Using PGI Compilers" in the PGI User's Guide.

Note

The C/C++ pragmas to which this section refers are not available in PVF.

OpenMP Overview

Let's look at the OpenMP shared-memory parallel programming model and some common OpenMP terminology.

OpenMP Shared-Memory Parallel Programming Model

The OpenMP shared-memory programming model is a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in Fortran, C and C++ programs.

Fortran directives
: Allow users to place hints in the source code to help the compiler generate more efficient code. You typically use directives to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. You place them in your source code where you want them to take effect; and they usually stay in effect from the point where included until the end of the compilation unit or until another directive or C/C++ pragma changes its status.

  Fortran directives and C/C++ pragmas include a parallel region construct for writing coarse grain SPMD programs, work-sharing constructs which specify that DO loop iterations should be split among the available threads of execution, and synchronization constructs.

  Note

  The data environment is controlled either by using clauses on the directives or with additional directives.

Runtime library routines
: Are available to query the parallel runtime environment, for example to determine how many threads are participating in execution of a parallel region.

Environment variables
: Are available to control the execution behavior of parallel programs. For more information on OpenMP, see *www.openmp.org*.

Terminology

For OpenMP 3.0 there are a number of terms for which it is useful to have common definitions.

Thread
: An execution entity with a stack and associated static memory, called *threadprivate memory*.

  - An OpenMP thread is a thread that is managed by the OpenMP runtime system.
  - A thread-safe routine is a routine that performs the intended function even when executed concurrently, that is, by more than one thread.

Region
: All code encountered during a specific instance of the execution of a given construct or of an OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.

  Regions are *nested* if one region is (dynamically) enclosed by another region, that is, a region is encountered during the execution of another region. PGI currently does not support nested parallel regions.

Parallel region
: In OpenMP 3.0 there is a distinction between a parallel region and an active parallel region. A parallel region can be either inactive or active.

  - An inactive parallel region is executed by a single thread.

- An active parallel region is a parallel region that is executed by a team consisting of more than one thread.

  Note

  The definition of an active parallel region changed between OpenMP 2.5 and OpenMP 3.0. In OpenMP 2.5, the definition was a parallel region whose IF clause evaluates to true. To examine the significance of this change, look at the following example:

  ```
       program test
       logical omp_in_parallel

  !$omp parallel
       print *, omp_in_parallel()
  !$omp end parallel

       stop
       end
  ```

  Suppose we run this program with OMP_NUM_THREADS set to one. In OpenMP 2.5, this program yields T while in OpenMP 3.0, the program yields F. In OpenMP 3.0, execution is not occurring by more than one thread. Therefore, change in this definition may mean previous programs require modification.

Task
: A specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct.

OpenMP Example

Look at the following simple OpenMP example involving loops.

Example 8.1. OpenMP Loop Example

```
      PROGRAM MAIN
      INTEGER I, N, OMP_GET_THREAD_NUM
      REAL*8 V(1000), GSUM, LSUM

      GSUM = 0.0D0
      N = 1000

      DO I = 1, N
         V(I) = DBLE(I)
      ENDDO

!$OMP PARALLEL PRIVATE(I,LSUM) SHARED(V,GSUM,N)
      LSUM = 0.0D0
!$OMP DO
      DO I = 1, N
         LSUM = LSUM + V(I)
      ENDDO
!$OMP END DO
!$OMP CRITICAL
      print *, "Thread ",OMP_GET_THREAD_NUM()," local sum: ",LSUM
      GSUM = GSUM + LSUM
!$OMP END CRITICAL
!$OMP END PARALLEL
```

```
      PRINT *, "Global Sum: ",GSUM

      STOP
      END
```

If you execute this example with the environment variable OMP_NUM_THREADS set to 4, then the output looks similar to this:

```
Thread            0 local sum:    31375.00000000000
Thread            1 local sum:    93875.00000000000
Thread            2 local sum:    156375.0000000000
Thread            3 local sum:    218875.0000000000
Global Sum:    500500.0000000000
FORTRAN STOP
```

Task Overview

Every part of an OpenMP program is part of a task. A task, whose execution can be performed immediately or delayed, has these characteristics:

- Code to execute
- A data environment - that is, it owns its data
- An assigned thread that executes the code and uses the data.

There are two activities associated with tasks: packaging and execution.

- Packaging: Each encountering thread packages a new instance of a task - code and data.
- Execution: Some thread in the team executes the task at some later time.

In the following sections, we use this terminology:

Task
: The package of code and instructions for allocating data created when a thread encounters a task construct. A task can be implicit or explicit.

  - An explicit task is a task generated when a task construct is encountered during execution.
  - An implicit task is a task generated by the implicit parallel region or generated when a parallel construct is encountered during execution.

Task construct
: A task directive plus a structured block

Task region
: The dynamic sequence of instructions produced by the execution of a task by a thread.

Tasks

Every part of an OpenMP program is part of a task. "Task Overview," on page 154 provides a general overview of tasks and general terminology associated with tasks. This section provides more detailed information about tasks, including tasks scheduling points and the task construct.

Task Characteristics and Activities

A task, whose execution can be performed immediately or delayed, has these characteristics:

- Code to execute
- A data environment - that is, it owns its data
- An assigned thread that executes the code and uses the data.

There are two activities associated with tasks: packaging and execution.

- Packaging: Each encountering thread packages a new instance of a task - code and data.
- Execution: Some thread in the team executes the task at some later time.

Task Scheduling Points

PGI currently supports four task scheduling points: at the beginning of a task, at the end of a task, a taskwait, and at a barrier.

- Beginning of a task.

  At the beginning of a task, the task can be executed immediately or registered for later execution. A programmer-specified "if" clause that is FALSE forces immediate execution of the task. The implementation can also force immediate execution; for example, a task within a task is never registered for later execution, it executes immediately.

- End of a task

  At the end of a task, the behavior of the scheduling point depends on how the task was executed. If the task was immediately executed, execution continues to the next statement. If it was previously registered and is being executed "out of sequence", control returns to where the task was executed - a taskwait.

- Taskwait

  A taskwait executes all registered tasks at the time it is called. In addition to executing all tasks registered by the calling thread, it also executes tasks previously registered by other threads. Let's take a quick look at this process.

  Suppose thread 0 called taskwait and is executing tasks and that thread 1 is registering tasks. Depending on the timing between thread 0 and thread 1, thread 0 may execute none of the tasks, all of the tasks, or some of tasks.

  ### Note

  Taskwait waits only for immediate children tasks, not for descendant tasks. You can achieve waiting on descendants but ensuring that each child also waits on its children.

- Barrier

  A barrier can be explicit or implicit. An example of an implicit barrier is the end of a parallel region.

The barrier effectively contains taskwaits. All threads must arrive at the barrier for the barrier to complete. This rule guarantees that all tasks have been executed at the completion of the barrier.

Task Construct

A task construct is a task directive plus a structured block, with the following syntax:

```
#pragma omp task [clause[[,]clause] ...]
     structured-block
```

where clause can be one of the following:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none )
```

Consider the following simple example of a program using tasks. This example illustrates the difference between registering tasks and executing tasks, a concept that is fundamental to understanding tasks.

This program contains a parallel region that contains a single region. The single region contains a loop that registers 10 tasks. Before reading the explanation that follows the example, consider what happens if you use four threads with this example.

Example 8.2. OpenMP Task Fortran Example

```
     PROGRAM MAIN
     INTEGER I
     INTEGER omp_get_thread_num
!$OMP PARALLEL PRIVATE(I)
!$OMP SINGLE
     DO I = 1, 10
        CALL SLEEP(MOD(I,2))
        PRINT *,"TASK ",I," REGISTERED BY THREAD ",omp_get_thread_num()
!$OMP TASK FIRSTPRIVATE(I)
        CALL SLEEP(MOD(I,5))
        PRINT *,"TASK ",I," EXECUTED BY THREAD   ",omp_get_thread_num()
!$OMP END TASK
     ENDDO
!$OMP END SINGLE
!$OMP END PARALLEL
     END
```

If you run this program with four threads, 0 through 3, one thread is in the single region registering tasks. The other three threads are in the implied barrier at the end of the single region executing tasks. Further, when the thread executing the single region completes registering the tasks, it joins the other threads and executes tasks.

The program includes calls to `sleep` to slow the program and allow all threads to participate.

The output for the Fortran example is similar to the following. In this output, thread 1 was registering tasks while the other three threads - 0,2, and 3 - were executing tasks When all 10 tasks were registered, thread 1 began executing tasks as well.

```
TASK 1  REGISTERED BY THREAD 1
TASK 2  REGISTERED BY THREAD 1
TASK 1  EXECUTED BY THREAD 0
TASK 3  REGISTERED BY THREAD 1
TASK 4  REGISTERED BY THREAD 1
TASK 2  EXECUTED BY THREAD 3
TASK 5  REGISTERED BY THREAD 1
TASK 6  REGISTERED BY THREAD 1
TASK 6  EXECUTED BY THREAD 3
TASK 5  EXECUTED BY THREAD 3
TASK 7  REGISTERED BY THREAD 1
TASK 8  REGISTERED BY THREAD 1
TASK 3  EXECUTED BY THREAD 0
TASK 9  REGISTERED BY THREAD 1
TASK 10  REGISTERED BY THREAD 1
TASK 10  EXECUTED BY THREAD 1
TASK 4  EXECUTED BY THREAD 2
TASK 7  EXECUTED BY THREAD 0
TASK 8  EXECUTED BY THREAD 3
TASK 9  EXECUTED BY THREAD 1
```

Parallelization Directives

Parallelization directives are comments in a program that are interpreted by the PGI Fortran compilers when the option `–mp` is specified on the command line. The form of a parallelization directive is:

```
sentinel directive_name [clauses]
```

With the exception of the SGI-compatible DOACROSS directive, the *sentinel* must comply with these rules:

- Be one of these: !OMP, COMP, or *$OMP.
- Must start in column 1 (one).
- Must appear as a single word without embedded white space.
- The sentinel marking a DOACROSS directive is C$.

In addition to the sentinel rules, the directive must also comply with these rules:

- Standard Fortran syntax restrictions, such as line length, case insensitivity, and so on, apply to the directive line.
- Initial directive lines must have a space or zero in column six.
- Continuation directive lines must have a character other than a space or a zero in column six. Continuation lines for C$DOACROSS directives are specified using the C$& sentinel.
- Directives which are presented in pairs must be used in pairs.

Valid clauses depend on the directive. Clauses associated with directives have these characteristics:

- The order in which clauses appear in the parallelization directives is not significant.
- Commas separate clauses within the directives, but commas are not allowed between the directive name and the first clause.

- Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause.

Directive Recognition

The compiler option `–mp` enables recognition of the parallelization directives. The use of this option also implies:

`–Mreentrant`
: Local variables are placed on the stack and optimizations, such as `-Mnoframe`, that may result in non-reentrant code are disabled.

`–Miomutex`
: Critical sections are generated around Fortran I/O statements.

Many of the directives are presented in pairs and must be used in pairs. In the examples given with each section, the routines omp_get_num_threads() and omp_get_thread_num() are used; refer to Runtime Library Routines for more information. These routines return the number of threads currently in the team executing the parallel region and the thread number within the team, respectively.

Directive Clauses

Some directives accept clauses that further allow a user to control the scope attributes of variables for the duration of the directive or pragma. Not all clauses are allowed on all directives, so the clauses that are valid are included with the description of the directive. Typically, if no data scope clause is specified for variables, the default scope is *share*

The following table provides a brief summary of the clauses associated with OpenMP directives that PGI supports. Following the table is more detailed information about these clauses. For complete information on OpenMP and use of these clauses, refer to the User's Guide and to the OpenMP documentation available on the WorldWide Web.

Table 8.1. Directive Clauses Summary Table

Clause	Applies to	Description
COLLAPSE (n)	DO...END DO PARALLEL DO ... END PARALLEL DO PARALLEL WORKSHARE	Specifies how many loops are associated with the loop construct.
COPYIN (list)	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Allows threads to access the master thread's value, for a threadprivate variable. You assign the same value to threadprivate variables for each thread in the team executing the parallel region. Then, for each variable specified, the value of the variable in the master thread of the team is copied to the threadprivate copies at the beginning of the parallel region.

Clause	Applies to	Description
COPYPRIVATE(list)	END SINGLE	Specifies that one or more variables should be shared among all threads. This clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members.
DEFAULT	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables.
FIRSTPRIVATE(list)	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	Specifies that each thread should have its own instance of a variable, and that each variable in the list should be initialized with the value of the original variable, because it exists before the parallel construct.
IF()	PARALLEL ... END PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies whether a loop should be executed in parallel or in serial.
LASTPRIVATE(list)	DO PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS SECTIONS	Specifies that the enclosing context's version of the variable is set equal to the *private* version of whichever thread executes the final iteration of a for-loop construct.
NOWAIT	DO ... END DO SECTIONS SINGLE WORKSHARE . . . END WORKSHARE	Overrides the barrier implicit in a directive.

Clause	Applies to	Description
NUM_THREADS	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Sets the number of threads in a thread team.
ORDERED	DO...END DO PARALLEL DO ... END PARALLEL DO	Required on a parallel FOR statement if an ordered directive is used in the loop.
PRIVATE	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS SINGLE	Specifies that each thread should have its own instance of a variable.
REDUCTION({operator | intrinsic } : list)	DO PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE SECTIONS	Specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.
SCHEDULE(type [,chunk])	DO ... END DO PARALLEL DO... END PARALLEL DO	Applies to the FOR directive, allowing the user to specify the chunking method for parallelization. Work is assigned to threads in different manners depending on the scheduling type or chunk size used.
SHARED	PARALLEL PARALLEL DO ... END PARALLEL DO PARALLEL SECTIONS ... END PARALLEL SECTIONS PARALLEL WORKSHARE	Specifies that one or more variables should be shared among all threads. All threads within a team access the same storage area for shared variables
UNTIED	TASK TASKWAIT	Specifies that any thread in the team can resume the task region after a suspension.

COLLAPSE (n)

The COLLAPSE(n) clause specifies how many loops are associated with the loop construct.

The parameter of the collapse clause must be a constant positive integer expression. If no COLLAPSE clause is present, the only loop that is associated with the loop construct is the one that immediately follows the construct.

If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space, which is then divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

If the loop directive contains a COLLAPSE clause then there may be more than one associated loop.

COPYIN (list)

The COPYIN(list) clause allows threads to access the master thread's value, for a threadprivate variable. You assign the same value to threadprivate variables for each thread in the team executing the parallel region; that is, for each variable specified, the value of the variable in the master thread of the team is copied to the threadprivate copies at the beginning of the parallel region.

The COPYIN clause applies only to THREADPRIVATE common blocks. If you specify a COPYIN clause, here are a few tips:

- You cannot specify the same entity name more than once in the list.
- You cannot specify the same entity name in separate COPYIN clauses of the same directive.
- You cannot specify both a common block name and any variable within that same named common block in the list.
- You cannot specify both a common block name and any variable within that same named common block in separate COPYIN clauses of the same directive.

COPYPRIVATE(list)

The COPYPRIVATE(list) clause specifies that one or more variables should be shared among all threads. This clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members.

You use a COPYPRIVATE(list) clause on an END SINGLE directive to cause the variables in the list to be copied from the private copies in the single thread that executes the SINGLE region to the other copies in all other threads of the team at the end of the SINGLE region.

Note

The COPYPRIVATE clause must not appear on the same END SINGLE directive as a NOWAIT clause.

The compiler evaluates a COPYPRIVATE clause before any threads have passed the implied BARRIER directive at the end of that construct.

DEFAULT

The DEFAULT clause specifies the behavior of unscoped variables in a parallel region, such as the data-sharing attributes of variables. The DEFAULT clause lets you specify the default attribute for variables in the lexical extent of the parallel region. Individual clauses specifying PRIVATE, SHARED, and so on, override the declared DEFAULT.

Specifying DEFAULT(NONE) declares that there is no implicit default. With this declaration, each variable in the parallel region must be explicitly listed with an attribute of PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION.

FIRSTPRIVATE(list)

The FIRSTPRIVATE(list) clause specifies that each thread should have its own instance of a variable, and that each variable in the list should be initialized with the value of the original variable, because it exists before the parallel construct.

Variables that appear in the list of a FIRSTPRIVATE clause are subject to the same semantics as PRIVATE variables; however, these variables are initialized from the original object that exists prior to entering the parallel region.

If a directive construct contains a FIRSTPRIVATE argument to a Message Passing Interface (MPI) routine performing non-blocking communication, the MPI communication must complete before the end of the construct.

IF()

The IF() clause specifies whether a loop should be executed in parallel or in serial.

In the presence of an IF clause, the parallel region is executed in parallel only if the corresponding `scalar_logical_expression` evaluates to .TRUE.. Otherwise, the code within the region is executed by a single processor, regardless of the value of the environment variable `OMP_NUM_THREADS`.

LASTPRIVATE(list)

The LASTPRIVATE(list) clause specifies that the enclosing context's version of the variable is set equal to the *private* version of whichever thread executes the final iteration (for-loop construct).

NOWAIT

The NOWAIT clause overrides the barrier implicit in a directive. When you specify NOWAIT, it removes the implicit barrier synchronization at the end of a for or sections construct.

NUM_THREADS

The NUM_THREADS clause sets the number of threads in a thread team. The num_threads clause allows a user to request a specific number of threads for a parallel construct. If the num_threads clause is present, then

ORDERED

The ORDERED clause specifies that a loop is executed in the order of the loop iterations. This clause is required on a parallel FOR statement when an ordered directive is used in the loop.

You use this clause in conjunction with a DO or SECTIONS construct to impose a serial order on the execution of a section of code. If ORDERED constructs are contained in the dynamic extent of the DO construct, the ordered clause must be present on the DO directive.

PRIVATE

The PRIVATE clause specifies that each thread should have its own instance of a variable. Therefore, variables specified in a PRIVATE list are private to each thread in a team. In effect, the compiler creates a separate copy of each of these variables for each thread in the team. When an assignment to a private variable occurs, each thread assigns to its local copy of the variable. When operations involving a private variable occur, each thread performs the operations using its local copy of the variable.

Tips about private variables:

- Variables declared private in a parallel region are undefined upon entry to the parallel region. If the first use of a private variable within the parallel region is in a right-hand-side expression, the results of the expression will be undefined, indicating the probability of a coding error.
- Variables declared private in a parallel region are undefined when serial execution resumes at the end of the parallel region.

REDUCTION

The REDUCTION clause specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region. updates named variables declared on the clause within the directive construct.

Intermediate values of REDUCTION variables are not used within the parallel construct, other than in the updates themselves.Variables that appear in the list of a REDUCTION clause must be SHARED. A private copy of each variable in `list` is created for each thread as if the PRIVATE clause had been specified. Each private copy is initialized according to the operator as specified in the following table:

Table 8.2. Initialization of REDUCTION Variables

Operator / Intrinsic	Initialization
+	0
*	1
-	0
.AND.	.TRUE.

Operator / Intrinsic	Initialization
.NEQV.	.FALSE.
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on

Operator / Intrinsic	Initialization
.OR.	.FALSE.
.EQV.	.TRUE.

Operator / Intrinsic	Initialization
IOR	0
IEOR	0

At the end of the parallel region, a reduction is performed on the instances of variables appearing in `list` using operator or intrinsic as specified in the REDUCTION clause. The initial value of each REDUCTION variable is included in the reduction operation. If the `{operator | intrinsic}:` portion of the REDUCTION clause is omitted, the default reduction operator is "+" (addition).

SCHEDULE

The SCHEDULE clause specifies how iterations of the DO loop are divided up between processors. Given a SCHEDULE (type [, chunk]) clause, the type can be STATIC, DYNAMIC, GUIDED, or RUNTIME, defined in the following list.

- When SCHEDULE (STATIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. The blocks of iterations are statically assigned to threads in a round-robin fashion in order of the thread ID numbers. The chunk must be a scalar integer expression. If chunk is not specified, a default chunk size is chosen equal to:

```
(number_of_iterations + omp_num_threads() - 1) / omp_num_threads()
```

- When SCHEDULE (DYNAMIC, chunk) is specified, iterations are allocated in contiguous blocks of size chunk. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. The chunk must be a scalar integer expression. If no chunk is specified, a default chunk size is chosen equal to 1.
- When SCHEDULE (GUIDED, chunk) is specified, the chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space. Chunk specifies the minimum number of iterations to dispatch each time, except when there are less than chunk iterations remaining to be processed, at which point all remaining iterations are assigned. If no chunk is specified, a default chunk size is chosen equal to 1.
- When SCHEDULE (RUNTIME) is specified, the decision regarding iteration scheduling is deferred until runtime. The schedule type and chunk size can be chosen at runtime by setting the OMP_SCHEDULE environment variable. If this environment variable is not set, the resulting schedule is equivalent to SCHEDULE(STATIC).

SHARED

The SHARED clause specifies variables that must be available to all threads. If you specify a variable as SHARED, you are stating that all threads can safely share a single copy of the variable. When one or more variables are shared among all threads, all threads access the same storage area for the shared variables.

UNTIED

The UNTIED clause specifies that any thread in the team can resume the task region after a suspension.

Note

The thread number may change at any time during the execution of an untied task. Therefore, the value returned by `omp_get_thread_num` is generally not useful during execution of such a task region.

Directive Summary Table

Table 8.3 provides a brief summary of the directives and pragmas that PGI supports.

Table 8.3. Directive Summary Table

Directive	Description
ATOMIC	Semantically equivalent to enclosing a single statement in the CRITCIAL...END CRITICAL directive. Note: Only certain statements are allowed.
BARRIER	Synchronizes all threads at a specific point in a program so that all threads complete work to that point before any thread continues.
CRITICAL ... END CRITICAL	Defines a subsection of code within a parallel region, a critical section, which is executed one thread at a time.
C$DOACROSS	Provides a mechanism for distribution of loop iterations across the available threads in a parallel region.
DO...END DO	Specifies that the compiler should parallelize the loop to which it applies, even though that loop is not contained within a parallel region.
FLUSH	When this appears, all processor-visible data items, or, when a list is present (FLUSH [list]), only those specified in the list, are written to memory, thus ensuring that all the threads in a team have a consistent view of certain objects in memory.
MASTER ... END MASTER	Designates code that executes on the master thread and that is skipped by the other threads.
ORDERED	Defines a code block that is executed by only one thread at a time, and in the order of the loop iterations; this makes the ordered code block sequential, while allowing parallel execution of statements outside the code block.
PARALLEL ... END PARALLEL	Supports a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.
PARALLEL DO	Enables you to specify which loops the compiler should parallelize.
PARALLEL SECTIONS	Defines a non-iterative work-sharing construct without the need to define an enclosing parallel region.
PARALLEL WORKSHARE	Provides a short form method for including a WORKSHARE directive inside a PARALLEL construct.

Directive	Description
SECTIONS … END SECTIONS	Defines a non-iterative work-sharing construct within a parallel region.
SINGLE ... END SINGLE	Designates code that executes on a single thread and that is skipped by the other threads.
TASK	Defines an explicit task.
TASKWAIT	Specifies a wait on the completion of child tasks generated since the beginning of the current task.
THREADPRIVATE	When a common block or variable that is initialized appears in this directive, each thread's copy is initialized once prior to its first use.
WORKSHARE ... END WORKSHARE	Provides a mechanism to effect parallel execution of non-iterative but implicitly data parallel constructs.

ATOMIC

The OpenMP ATOMIC directive is semantically equivalent to a single statement in a CRITICAL...END CRITICAL directive.

Syntax:

```
!$OMP ATOMIC
```

Usage:

The ATOMIC directive is semantically equivalent to enclosing the following single statement in a CRITICAL / END CRITICAL directive pair.

The statements must be one of the following forms:

```
x = x operator expr
```

```
x = expr operator x
```

```
x = intrinsic (x, expr)
```

```
x = intrinsic (expr, x)
```

where `x` is a scalar variable of intrinsic type, `expr` is a scalar expression that does not reference `x`, intrinsic is one of MAX, MIN, IAND, IOR, or IEOR, and operator is one of +, *, -, /, .AND., .OR., .EQV., or .NEQV..

BARRIER

The OpenMP BARRIER directive defines a point in a program where each thread waits for all other threads to arrive before continuing with program execution.

Syntax:

```
!$OMP BARRIER
```

Usage:

There may be occasions in a parallel region when it is necessary that all threads complete work to that point before any thread is allowed to continue. The BARRIER directive synchronizes all threads at such a point in a program. Multiple barrier points are allowed within a parallel region. The BARRIER directive must either be executed by all threads executing the parallel region or by none of them.

CRITICAL ... END CRITICAL

The CRITICAL...END CRITICAL directive requires a thread to wait until no other thread is executing within a critical section.

Syntax:

```
!$OMP CRITICAL [(name)]
  < Fortran code executed in body of critical section >
!$OMP END CRITICAL [(name)]
```

Usage:

Within a parallel region, there may exist subregions of code that will not execute properly when executed by multiple threads simultaneously. This issue is often due to a shared variable that is written and then read again.

The CRITICAL... END CRITICAL directive pair defines a subsection of code within a parallel region, referred to as a critical section, which is executed one thread at a time.

The first thread to arrive at a critical section is the first to execute the code within the section. The second thread to arrive does not begin execution of statements in the critical section until the first thread exits the critical section. Likewise, each of the remaining threads wait its turn to execute the statements in the critical section.

You can use the optional name argument to identify the critical region. Names that identify critical regions have external linkage and are in a name space separate from the name spaces used by labels, tags, members, and ordinary identifiers. If a name argument appears on a CRITICAL directive, the same name must appear on the END CRITICAL directive.

Note

Critical sections cannot be nested, and any such specifications are ignored. Branching into or out of a critical section is illegal.

Example of Critical...End Critical directive:

```
 PROGRAM CRITICAL_USE
 REAL A(100,100),MX, LMX
 INTEGER I, J
 MX = -1.0
 LMX = -1.0
 CALL RANDOM_SEED()
 CALL RANDOM_NUMBER(A)
!$OMP PARALLEL PRIVATE(I), FIRSTPRIVATE(LMX)
!$OMP DO
  DO J=1,100
   DO I=1,100
    LMX = MAX(A(I,J),LMX)
   ENDDO
  ENDDO
!$OMP CRITICAL
  MX = MAX(MX,LMX)
!$OMP END CRITICAL
!$OMP END PARALLEL
  PRINT *,"MAX VALUE OF A IS ", MX
 END
```

This program could also be implemented without the critical region by declaring MX as a reduction variable and performing the MAX calculation in the loop using MX directly rather than using LMX. Refer to "PARALLEL ... END PARALLEL " and "DO...END DO " for more information on how to use the REDUCTION clause on a parallel DO loop.

C$DOACROSS

The C$DOACROSS directive, while not part of the OpenMP standard, is supported for compatibility with programs parallelized using legacy SGI-style directives.

Syntax:

```
C$DOACROSS [ Clauses ]
  < Fortran DO loop to be executed in parallel >
```

Clauses:

{PRIVATE | LOCAL} (list)
{SHARED | SHARE} (list)
MP_SCHEDTYPE={SIMPLE | INTERLEAVE}
CHUNK=<integer_expression>
IF (logical_expression)

Usage:

The C$DOACROSS directive has the effect of a combined parallel region and parallel DO loop applied to the loop immediately following the directive. It is very similar to the OpenMP PARALLEL DO directive, but provides for backward compatibility with codes parallelized for SGI systems prior to the OpenMP standardization effort.

The C$DOACROSS directive must not appear within a parallel region. It is a shorthand notation that tells the compiler to parallelize the loop to which it applies, even though that loop is not contained within a parallel region.

Important

While The C$DOACROSS syntax may be more convenient, if multiple successive DO loops are to be parallelized, it is more efficient to define a single enclosing parallel region and parallelize each loop using the OpenMP DO directive.

A variable declared PRIVATE or LOCAL to a C$DOACROSS loop is treated the same as a private variable in a parallel region or DO. A variable declared SHARED or SHARE to a C$DOACROSS loop is shared among the threads, meaning that only 1 copy of the variable exists to be used and/or modified by all of the threads. This is equivalent to the default status of a variable that is not listed as PRIVATE in a parallel region or DO. This same default status is used in C$DOACROSS loops as well.

For more information on clauses, refer to "Directive Clauses," on page 158.

DO...END DO

The OpenMP DO...END DO directive support parallel execution and the distribution of loop iterations across available threads in a parallel region.

Syntax:

```
!$OMP DO [Clauses]
  < Fortran DO loop to be executed in parallel>
!$OMP END DO [NOWAIT]
```

Clauses:

PRIVATE(list)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)
SCHEDULE (type [, chunk])
COLLAPSE (n)
ORDERED

Usage:

The real purpose of supporting parallel execution is the distribution of work across the available threads. The DO... END DO directive pair provides a convenient mechanism for the distribution of loop iterations across the available threads in a parallel region.

While you can explicitly manage work distribution with constructs such as the following one, these constructs are not in the form of directives.

Examples:

```
IF (omp_get_thread_num() .EQ. 0)
THEN
  ...
ELSE IF (omp_get_thread_num() .EQ. 1)
THEN
   ...
ENDIF
```

Tips

Remember these items about clauses in the DO...END DO directives:

- Variables declared in a PRIVATE list are treated as private to each thread participating in parallel execution of the loop, meaning that a separate copy of the variable exists with each thread.
- Variables declared in a FIRSTPRIVATE list are PRIVATE, and are initialized from the original object existing before the construct.
- Variables declared in a LASTPRIVATE list are PRIVATE, and the thread that executes the sequentially last iteration updates the version of the object that existed before the construct.
- The REDUCTION clause for the directive is described in “Directive Clauses,” on page 158.
- The SCHEDULE clause specifies how iterations of the DO loop are divided up between threads. For more information on this clause, refer to “Directive Clauses,” on page 158.
- If ORDERED code blocks are contained in the dynamic extent of the DO directive, the ORDERED clause must be present. For more information on ORDERED code blocks, refer to “ORDERED ”.
- The DO... END DO directive pair directs the compiler to distribute the iterative DO loop immediately following the !$OMP DO directive across the threads available to the program. The DO loop is executed in parallel by the team that was started by an enclosing parallel region. If the !$OMP END DO directive is not specified, the !$OMP DO is assumed to end with the enclosed DO loop. DO... END DO directive pairs may not be nested. Branching into or out of a !$OMP DO loop is not supported.

- By default, there is an implicit barrier after the end of the parallel loop; the first thread to complete its portion of the work waits until the other threads have finished their portion of work. If NOWAIT is specified, the threads will not synchronize at the end of the parallel loop.

In addition to the preceding items, remember these items about !$OMP DO loops :

- The DO loop index variable is always private.
- !$OMP DO loops must be executed by all threads participating in the parallel region or none at all.
- The END DO directive is optional, but if it is present it must appear immediately after the end of the enclosed DO loop.
- Values of the loop control expressions and the chunk expressions must be the same for all threads executing the loop.

Example:

```
PROGRAM DO_USE
   REAL A(1000), B(1000)
   DO I=1,1000
      B(I) = FLOAT(I)
   ENDDO
!$OMP PARALLEL
!$OMP DO
   DO I=1,1000
      A(I) = SQRT(B(I));
   ENDDO
  ...
!$OMP END PARALLEL
   ...
   END
```

FLUSH

The OpenMP FLUSH directive ensures that processor-visible data item are written back to memory at the point at which the directive appears.

Syntax:

```
!$OMP FLUSH [(list)]
```

Usage:

The OpenMP FLUSH directive ensures that all processor-visible data items, or only those specified in `list`, when it is present, are written back to memory at the point at which the directive appears.

MASTER ... END MASTER

The MASTER...END MASTER directive allows the user to designate code that must execute on a master thread and that is skipped by other threads in the team of threads.

Syntax:

```
!$OMP MASTER
  < Fortran code executed in body of MASTER section >
!$OMP END MASTER
```

Usage:

A master thread is a single thread of control that begins an OpenMP program and which is present for the duration of the program. In a parallel region of code, there may be a sub-region of code that should execute only on the master thread. Instead of ending the parallel region before this subregion and then starting it up again after this subregion, the MASTER... END MASTER directive pair allows the user to conveniently designate code that executes on the master thread and is skipped by the other threads.

- There is no implied barrier on entry to or exit from a master section of code.
- Nested master sections are ignored.
- Branching into or out of a master section is not supported.

Examples:

Example of Fortran **MASTER...END MASTER** directive

```
PROGRAM MASTER_USE
     INTEGER A(0:1)
     INTEGER omp_get_thread_num
     A=-1
!$OMP PARALLEL
     A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP MASTER
     PRINT *, "YOU SHOULD ONLY SEE THIS ONCE"
!$OMP END MASTER
!$OMP END PARALLEL
     PRINT *, "A(0)=", A(0), " A(1)=", A(1)
 END
```

ORDERED

The OpenMP ORDERED directive allows the user to identify a portion of code within an ordered code block that must be executed in the original, sequential order, while allowing parallel execution of statements outside the code block.

Syntax:

```
!$OMP ORDERED
  < Fortran code block executed by processor >
!$OMP END ORDERED
```

Usage:

The ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive that includes the ORDERED clause. The structured code block between the ORDERED / END ORDERED directives is executed by only one thread at a time, and in the order of the loop iterations. This sequentializes the ordered code block while allowing parallel execution of statements outside the code block. The following additional restrictions apply to the ORDERED directive:

- The ordered code block must be a structured block.

- It is illegal to branch into or out of the block.
- A given iteration of a loop with a DO directive cannot execute the same ORDERED directive more than once, and cannot execute more than one ORDERED directive.

PARALLEL ... END PARALLEL

The OpenMP PARALLEL...END PARALLEL directive supports a fork/join execution model in which a single thread executes all statements until a parallel region is encountered.

Syntax:

```
!$OMP PARALLEL [Clauses]
  < Fortran code executed in body of parallel region >
!$OMP END PARALLEL
```

Clauses:

PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
REDUCTION([{operator | intrinsic}:] list)
COPYIN(list)
IF(scalar_logical_expression)
NUM_THREADS(scalar_integer_expression)

Usage:

This directive pair declares a region of parallel execution. It directs the compiler to create an executable in which the statements within the structured block, such as between PARALLEL and PARALLEL END for directives, are executed by multiple lightweight threads. The code that lies within this structured block is called a *parallel region*.

The OpenMP parallelization directives support a fork/join execution model in which a single thread executes all statements until a parallel region is encountered. At the entrance to the parallel region, a system-dependent number of symmetric parallel threads begin executing all statements in the parallel region redundantly. These threads share work by means of work-sharing constructs such as parallel DO loops or FOR loops.

- The number of threads in the team is controlled by the `OMP_NUM_THREADS` environment variable. If `OMP_NUM_THREADS` is not defined, the program executes parallel regions using only one processor.
- Branching into or out of a parallel region is not supported.
- All other shared-memory parallelization directives must occur within the scope of a parallel region. Nested PARALLEL... END PARALLEL directive pairs are not supported and are ignored.
- There is an implicit barrier at the end of the parallel region, which, in the directive, is denoted by the END PARALLEL directive. When all threads have completed execution of the parallel region, a single thread resumes execution of the statements that follow.

Note

By default, there is no work distribution in a parallel region. Each active thread executes the entire region redundantly until it encounters a directive that specifies work distribution. For work distribution, see the “DO...END DO ”, “PARALLEL DO ”, or “C$DOACROSS ” directives.

Example

Example of Fortran **PARALLEL...END PARALLEL** directive

```
 PROGRAM WHICH_PROCESSOR_AM_I
   INTEGER A(0:1)
   INTEGER omp_get_thread_num
   A(0) = -1
   A(1) = -1
!$OMP PARALLEL
   A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP END PARALLEL
   PRINT *, "A(0)=",A(0)," A(1)=",A(1)
 END
```

Clause Usage:

COPYIN: The COPYIN clause applies only to THREADPRIVATE common blocks. In the presence of the COPYIN clause, data from the master thread's copy of the common block is copied to the THREADPRIVATE copies upon entry to the parallel region.

IF: In the presence of an IF clause, the parallel region is executed in parallel only if the corresponding `scalar_logical_expression` evaluates to .TRUE.. Otherwise, the code within the region is executed by a single processor, regardless of the value of the environment variable `OMP_NUM_THREADS`.

NUM_THREADS: If the NUM_THREADS clause is present, the corresponding expression, `scalar_integer_expression`, must evaluate to a positive integer value. This value sets the maximum number of threads used during execution of the parallel region. A NUM_THREADS clause overrides either a previous call to the library routine omp_set_num_threads() or the setting of the OMP_NUM_THREADS environment variable.

PARALLEL DO

The OpenMP PARALLEL DO directive is a shortcut for a PARALLEL region that contains a single DO directive.

Note

The OpenMP PARALLEL DO or DO directive must be immediately followed by a DO statement (as defined by R818 of the ANSI Fortran standard). If you place another statement or an OpenMP directive between the PARALLEL DO or DO directive and the DO statement, the compiler issues a syntax error.

Syntax:

```
!$OMP PARALLEL DO [CLAUSES]
  < Fortran DO loop to be executed in parallel >
[!$OMP END PARALLEL DO]
```

Clauses:

PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION([{operator | intrinsic}:] list)
COPYIN(list)
IF(scalar_logical_expression)
NUM_THREADS(scalar_integer_expression)
SCHEDULE (type [, chunk])
COLLAPSE (n)
ORDERED

Usage:

The semantics of the PARALLEL DO directive are identical to those of a parallel region containing only a single parallel DO loop and directive. The available clauses are the same as those defined in “PARALLEL ... END PARALLEL ,” on page 172 and “DO...END DO ”.

Note

The END PARALLEL DO directive is optional.

PARALLEL SECTIONS

The OpenMP PARALLEL SECTIONS / END SECTIONS directive pair define tasks to be executed in parallel; that is, they define a non-iterative work-sharing construct without the need to define an enclosing parallel region.

Syntax:

```
!$OMP PARALLEL SECTIONS [CLAUSES]
[!$OMP SECTION]
    < Fortran code block executed by processor i >
[!$OMP SECTION]
    < Fortran code block executed by processor j >
 ...
!$OMP END SECTIONS [NOWAIT]
```

Clauses:

PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)
COPYIN (list)
IF(scalar_logical_expression)
NUM_THREADS(scalar_integer_expression)

Usage:

The PARALLEL SECTIONS / END SECTIONS directive pair define a non-iterative work-sharing construct without the need to define an enclosing parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors will have no work and will jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors will execute more than one section.

A SECTION directive may only appear within the lexical extent of the enclosing PARALLEL SECTIONS / END SECTIONS directives. In addition, the code within the PARALLEL SECTIONS / END SECTIONS directives must be a structured block, and the code in each SECTION must be a structured block.

Semantics are identical to a parallel region containing only an omp sections pragma and the associated structured block. The available clauses are as defined in “PARALLEL ... END PARALLEL ,” on page 172 and “DO...END DO ”.

PARALLEL WORKSHARE

The OpenMP PARALLEL WORKSHARE directive provides a short form method of including a WORKSHARE directive inside a PARALLEL construct.

Syntax:

```
!$OMP PARALLEL WORKSHARE [CLAUSES]
< Fortran structured block to be executed in parallel >
[!$OMP END PARALLEL WORKSHARE]
```

```
!$OMP PARALLEL DO [CLAUSES]
< Fortran DO loop to be executed in parallel >
[!$OMP END PARALLEL DO]
```

Clauses:

PRIVATE(list)
SHARED(list)
DEFAULT(PRIVATE | SHARED | NONE)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)
COPYIN (list)
IF(scalar_logical_expression)
NUM_THREADS(scalar_integer_expression)
SCHEDULE (type [, chunk])
COLLAPSE (n)
ORDERED

Usage:

The OpenMP PARALLEL WORKSHARE directive provides a short form method of including a WORKSHARE directive inside a PARALLEL construct. The semantics of the PARALLEL WORKSHARE directive are identical to those of a parallel region containing a single WORKSHARE construct.

The END PARALLEL WORKSHARE directive is optional, and NOWAIT may not be specified on an END PARALLEL WORKSHARE directive. The available clauses are as defined in "PARALLEL ... END PARALLEL ," on page 172.

SECTIONS ... END SECTIONS

The OpenMP SECTIONS / END SECTIONS directive pair define a non-iterative work-sharing construct within a parallel region in which each section is executed by a single processor.

Syntax:

```
!$OMP SECTIONS [ Clauses ]
[!$OMP SECTION]
    < Fortran code block executed by processor i >
[!$OMP SECTION]
    < Fortran code block executed by processor j >
 ...
!$OMP END SECTIONS [NOWAIT]
```

Clauses:

PRIVATE(list)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION({operator | intrinsic} : list)

Usage:

The SECTIONS / END SECTIONS directive pair defines a non-iterative work-sharing construct within a parallel region. Each section is executed by a single processor. If there are more processors than sections, some processors have no work and thus jump to the implied barrier at the end of the construct. If there are more sections than processors, one or more processors must execute more than one section.

A SECTION directive may only appear within the lexical extent of the enclosing SECTIONS / END SECTIONS directives. In addition, the code within the SECTIONS / END SECTIONS directives must be a structured block.

The available clauses are as defined in "PARALLEL ... END PARALLEL ," on page 172 and "DO...END DO ".

SINGLE ... END SINGLE

The SINGLE...END SINGLE directive designates code that executes on a single thread and that is skipped by the other threads.

Syntax:

```
!$OMP SINGLE [Clauses]
  < Fortran code executed in body of SINGLE processor section >
!$OMP END SINGLE [NOWAIT]
```

Clauses:

PRIVATE(list)
FIRSTPRIVATE(list)
COPYPRIVATE(list)

Usage:

In a parallel region of code, there may be a sub-region of code that only executes correctly on a single thread. Instead of ending the parallel region before this subregion and then starting it up again after this subregion, the SINGLE...END SINGLE directive pair or the omp single pragma lets you conveniently designate code that executes on a single thread and is skipped by the other threads.

The following restrictions apply to the SINGLE...END SINGLE directive:

- There is an implied barrier on exit from a SINGLE...END SINGLE section of code unless the optional NOWAIT clause is specified.
- Nested single process sections are ignored.
- Branching into or out of a single process section is not supported.

Examples:

```
 PROGRAM SINGLE_USE
   INTEGER A(0:1)
   INTEGER omp_get_thread_num()
!$OMP PARALLEL
   A(omp_get_thread_num()) = omp_get_thread_num()
!$OMP SINGLE
   PRINT *, "YOU SHOULD ONLY SEE THIS ONCE"
!$OMP END SINGLE
!$OMP END PARALLEL
   PRINT *, "A(0)=",A(0), " A(1)=", A(1)
 END
```

TASK

The OpenMP TASK directive defines an explicit task.

Syntax:

```
!$OMP TASK [Clauses]
   < Fortran code executed as task>
!$OMP END TASK
```

Clauses:

IF(scalar_logical_expression)	PRIVATE(list)
UNTIED	FIRSTPRIVATE(list)
DEFAULT(private | firstprivate | shared | none)	SHARED(list)

Usage:

The TASK / END TASK directive pair defines an explicit task.

When a thread encounters a task construct, a task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply. The encountering thread may immediately execute the task, or delay its execution. If the task execution is delayed, then any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs.

A task construct may be nested inside an outer task, but the task region of the inner task is not a part of the task region of the outer task.

When an if clause is present on a task construct and the if clause expression evaluates to false, the encountering thread must suspend the current task region and begin execution of the generated task immediately, and the suspended task region may not be resumed until the generated task is completed. The task still behaves as a distinct task region with respect to data environment, lock ownership, and synchronization constructs.

Note

Use of a variable in an if clause expression of a task construct causes an implicit reference to the variable in all enclosing constructs.

A thread that encounters a task scheduling point within the task region may temporarily suspend the task region. By default, a task is tied and its suspended task region can only be resumed by the thread that started its execution. If the untied clause is present on a task construct, any thread in the team can resume the task region after a suspension.

The task construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit task region includes a task scheduling point at its point of completion. An implementation may add task scheduling points anywhere in untied task regions.

Note

When storage is shared by an explicit task region, it is the programmer's responsibility to ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit task region completes its execution.

Restrictions:

The following restrictions apply to the TASK directive:

- A program that branches into or out of a task region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the task directive, or on any side effects of the evaluations of the clauses.
- At most one *if* clause can appear on the directive.
- Unsynchronized use of Fortran I/O statements by multiple tasks on the same unit has unspecified behavior.

TASKWAIT

The OpenMP TASKWAIT directive specifies a wait on the completion of child tasks generated since the beginning of the current task.

Syntax:

```
!$OMP TASKWAIT
```

Clauses:

IF(scalar_logical_expression)
UNTIED
DEFAULT(private | firstprivate | shared | none)
PRIVATE(list)
FIRSTPRIVATE(list)
SHARED(list)

Usage:

The OpenMP TASKWAIT directive specifies a wait on the completion of child tasks generated since the beginning of the current task.

Restrictions:

The following restrictions apply to the TASKWAIT directive:

- The TASKWAIT directive and the omp taskwait pragma may be placed only at a point where a base language statement is allowed.
- The taskwait directive may not be used in place of the statement following an *if*, *while*,*do*, *switch*, or *label*.

THREADPRIVATE

The OpenMP THREADPRIVATE directive identifies a Fortran common block as being private to each thread.

Syntax:

```
!$OMP THREADPRIVATE (list)
```

Usage:

The `list` for this directive is a comma-separated list of named variables to be made private to each thread or named common blocks to be made private to each thread but global within the thread.

On entry to a parallel region, data in a THREADPRIVATE common block or variable is undefined unless COPYIN is specified on the PARALLEL directive. When a common block or variable that is initialized using DATA statements appears in a THREADPRIVATE directive, each thread's copy is initialized once prior to its first use.

Restrictions:

The following restrictions apply to the THREADPRIVATE directive:

- The THREADPRIVATE directive must appear after every declaration of a thread private common block.
- Only named common blocks can be made thread private.
- Common block names must appear between slashes, such as `/common_block_name/`.
- This directive must appear in the declarations section of a program unit after the declaration of any common blocks or variables listed.
- It is illegal for a THREADPRIVATE common block or its constituent variables to appear in any clause other than a COPYIN clause.
- A variable can appear in a THREADRIVATE directive only in the scope in which it is declared. It must not be an element of a common block or be declared in an EQUIVALENCE statement.
- A variable that appears in a THREADPRIVATE directive and is not declared in the scope of a module must have the SAVE attribute.

WORKSHARE ... END WORKSHARE

The OpenMP WORKSHARE … END WORKSHARE directive pair provides a mechanism to effect parallel execution of non-iterative but implicitly data parallel constructs.

Syntax:

```
!$OMP WORKSHARE
   < Fortran structured block to be executed in parallel >
!$OMP END WORKSHARE [NOWAIT]
```

Usage:

The Fortran structured block enclosed by the WORKSHARE … END WORKSHARE directive pair can consist only of the following types of statements and constructs:

- Array assignments
- Scalar assignments
- FORALL statements or constructs
- WHERE statements or constructs
- OpenMP ATOMIC, CRITICAL or PARALLEL constructs

The work implied by these statements and constructs is split up between the threads executing the WORKSHARE construct in a way that is guaranteed to maintain standard Fortran semantics. The goal of the WORKSHARE construct is to effect parallel execution of non-iterative but implicitly data parallel array assignments, FORALL, and WHERE statements and constructs intrinsic to the Fortran language beginning with Fortran 90. The Fortran structured block contained within a WORKSHARE construct must not contain any user-defined function calls unless the function is ELEMENTAL.

Runtime Library Routines

User-callable functions are available to the programmer to query and alter the parallel execution environment.

Note

Unlimited OpenMP thread counts are available in all PGI configurations. The number of threads is unlicensed in the OpenMP runtime libraries - up to the hard limit of 64 threads.

The following table summarizes the runtime library calls, providing an example for each.

Table 8.4. Runtime Library Routines Summary

Runtime Library Routines with Examples
omp_get_num_threads Returns the number of threads in the team executing the parallel region from which it is called. When called from a serial region, this function returns 1. A nested parallel region is the same as a single parallel region. By default, the value returned by this function is equal to the value of the environment variable `OMP_NUM_THREADS` or to the value set by the last previous call to **omp_set_num_threads()**. `integer function omp_get_num_threads();`
omp_set_num_threads Sets the number of threads to use for the next parallel region. This subroutine can only be called from a serial region of code. If it is called from within a parallel region, or from within a subroutine that is called from within a parallel region, the results are undefined. Further, this subroutine has precedence over the `OMP_NUM_THREADS` environment variable. `subroutine omp_set_num_threads(scalar_integer_exp);`
omp_get_thread_num Returns the thread number within the team. The thread number lies between 0 and **omp_get_num_threads()-1**. When called from a serial region, this function returns 0. A nested parallel region is the same as a single parallel region. `integer function omp_get_thread_num();`

omp_get_ancestor_thread_num
 	
Returns, for a given nested level of the current thread, the thread number of the ancestor.
 	
`integer function omp_get_ancestor_thread_num(level);`
`integer level`

Runtime Library Routines with Examples
omp_get_active_level Returns the number of enclosing active parallel regions enclosing the task that contains the call. PGI currently supports only one level of active parallel regions, so the return value currently is 1. `integer function omp_get_active_level();`
omp_get_level Returns the number of parallel regions enclosing the task that contains the call. `integer function omp_get_level();`
omp_get_max_threads Returns the maximum value that can be returned by calls to **omp_get_num_threads()**. If **omp_set_num_threads()** is used to change the number of processors, subsequent calls to **omp_get_max_threads()** return the new value. Further, this function returns the maximum value whether executing from a parallel or serial region of code. `integer function omp_get_max_threads();`
omp_get_num_procs Returns the number of processors that are available to the program. `integer function omp_get_num_procs();`

omp_get_stack_size
 	
Returns the value of the OpenMP internal control variable that specifies the size that is used to create a stack for a newly created thread.
 	
This value may *not* be the size of the stack of the current thread.
 	
`!omp_get_stack_size interface`
`function omp_get_stack_size ()`
`use omp_lib_kinds`
`integer ( kind=OMP_STACK_SIZE_KIND )`
`:: omp_get_stack_size`
`end function omp_get_stack_size`
`end interface`

omp_set_stack_size Changes the value of the OpenMP internal control variable that specifies the size to be used to create a stack for a newly created thread. The integer argument specifies the stack size in kilobytes. The size of the stack of the current thread cannot be changed. In the PGI implementation, all OpenMP or auto-parallelization threads are created just prior to the first parallel region; therefore, only calls to `omp_set_stack_size()` that occur prior to the first region have an effect. `subroutine omp_set_stack_size(integer(KIND=OMP_STACK_SIZE_KIND));`

Runtime Library Routines with Examples

omp_get_team_size
 	
Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs.
 	
`integer function omp_get_team_size (level)`
`integer level`

omp_in_parallel Returns whether or not the call is within a parallel region. Returns `.TRUE.` if called from within a parallel region and `.FALSE.` if called outside of a parallel region. When called from within a parallel region that is serialized, for example in the presence of an IF clause evaluating `.FALSE.`, the function returns `.FALSE.`. `logical function omp_in_parallel();`
omp_set_dynamic Allows automatic dynamic adjustment of the number of threads used for execution of parallel regions. This function is recognized, but currently has no effect. `subroutine omp_set_dynamic(scalar_logical_exp);`
omp_get_dynamic Allows the user to query whether automatic dynamic adjustment of the number of threads used for execution of parallel regions is enabled. This function is recognized, but currently always returns `.FALSE.`. `logical function omp_get_dynamic();`

omp_set_nested
 	
Allows enabling/disabling of nested parallel regions.
 	
This function is recognized, but currently has no effect.
 	
`subroutine omp_set_nested(nested);`
`  logical nested`

omp_get_nested Allows the user to query whether dynamic adjustment of the number of threads available for execution of parallel regions is enabled. This function is recognized, but currently always returns `.FALSE.`. `logical function omp_get_nested();`

Runtime Library Routines with Examples

omp_set_schedule
 	
Retrieve the value of the run_sched_var.
 	
`subroutine omp_set_schedule(kind, modifier)`
`  include 'omp_lib_kinds.h'`
`  integer (kind=omp_sched_kind) kind`
`  integer modifier`

omp_get_schedule
 	
Retrieve the value of the run_sched_var.
 	
`subroutine omp_get_schedule(kind, modifier)`
`  include 'omp_lib_kinds.h'`
`  integer (kind=omp_sched_kind) kind`
`  integer modifier`

omp_get_wtime Returns the elapsed wall clock time, in seconds, as a DOUBLE PRECISION value. Times returned are per-thread times, and are not necessarily globally consistent across all threads. `double precision function omp_get_wtime();`
omp_get_wtick Returns the resolution of omp_get_wtime(), in seconds, as a DOUBLE PRECISION value. `double precision function omp_get_wtick();`

omp_init_lock
 	
Initializes a lock associated with the variable lock for use in subsequent calls to lock routines. [Not available in PVF]
 	
The initial state of the lock is unlocked. If the variable is already associated with a lock, it is illegal to make a call to this routine.
 	
`subroutine omp_init_lock(lock)`
`  include 'omp_lib_kinds.h'`
`  integer(kind=omp_lock_kind) lock`

omp_destroy_lock
 	
Disassociates a lock associated with the variable.
 	
`subroutine omp_destroy_lock(lock)`
`  include 'omp_lib_kinds.h'`
`  integer(kind=omp_lock_kind) lock`

Runtime Library Routines with Examples

omp_set_lock
 	
Causes the calling thread to wait until the specified lock is available.
 	
The thread gains ownership of the lock when it is available. If the variable is not already associated with a lock, it is illegal to make a call to this routine.
 	
`subroutine omp_set_lock(lock)`
`  include 'omp_lib_kinds.h'`
`  integer(kind=omp_lock_kind) lock`

omp_unset_lock
 	
Causes the calling thread to release ownership of the lock associated with `integer_var`.
 	
If the variable is not already associated with a lock, it is illegal to make a call to this routine.
 	
`subroutine omp_unset_lock(lock)`
`  include 'omp_lib_kinds.h'`
`  integer(kind=omp_lock_kind) lock`

omp_test_lock
 	
Causes the calling thread to try to gain ownership of the lock associated with the variable.
 	
The function returns `.TRUE.` if the thread gains ownership of the lock; otherwise it returns `.FALSE.`. If the variable is not already associated with a lock, it is illegal to make a call to this routine.
 	
`logical function omp_test_lock(lock)`
`  include 'omp_lib_kinds.h'`
`  integer(kind=omp_lock_kind) lock`

OpenMP Environment Variables

OpenMP environment variables allow you to control the behavior of OpenMP programs. These environment variables allow you to set and pass information that can alter the behavior of directives.

Table 8.5, "OpenMP-related Environment Variable Summary Table," on page 184 provides a brief summary of these variables. After the table this section contains more information about each of them. For complete information and more details related to these environment variables, refer to the OpenMP documentation available on the WorldWide Web.

Table 8.5. OpenMP-related Environment Variable Summary Table

Environment Variable	Default	Description
OMP_DYNAMIC	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.
OMP_MAX_ACTIVE_LEVELS		Specifies the maximum number of nested parallel regions.
OMP_NESTED	FALSE	Currently has no effect. Typically enables (TRUE) or disables (FALSE) nested parallelism.

Environment Variable	Default	Description
OMP_NUM_THREADS	1	Specifies the number of threads to use during execution of parallel regions.
OMP_SCHEDULE	STATIC with chunk size of 1	Specifies the type of iteration scheduling and optionally the chunk size to use for *omp for* and *omp parallel for* loops that include the runtime schedule clause. The supported schedule types, which can be specified in upper- or lower-case are static, dynamic, guided, and auto.
OMP_STACKSIZE		Overrides the default stack size for a newly created thread.
OMP_THREAD_LIMIT	64	Specifies the absolute maximum number of threads that can be used in a program.
OMP_WAIT_POLICY	ACTIVE	Sets the behavior of idle threads, defining whether they spin or sleep when idle. The values are ACTIVE and PASSIVE.

OMP_DYNAMIC

`OMP_DYNAMIC` currently has no effect. Typically this variable enables (TRUE) or disables (FALSE) the dynamic adjustment of the number of threads.

OMP_MAX_ACTIVE_LEVELS

`OMP_MAX_ACTIVE_LEVELS` specifies the maximum number of nested parallel regions.

OMP_NESTED

`OMP_NESTED` currently has no effect. Typically this variable enables (TRUE) or disables (FALSE) nested parallelism.

OMP_NUM_THREADS

`OMP_NUM_THREADS` specifies the number of threads to use during execution of parallel regions. The default value for this variable is 1. For historical reasons, the environment variable `NCPUS` is supported with the same functionality. In the event that both `OMP_NUM_THREADS` and `NCPUS` are defined, the value of `OMP_NUM_THREADS` takes precedence.

Note

`OMP_NUM_THREADS` defines the threads that are used to execute the program, regardless of the number of physical processors available in the system. As a result, you can run programs using more threads than physical processors and they execute correctly. However, performance of programs executed in this manner can be unpredictable, and oftentimes will be inefficient.

OMP_SCHEDULE

`OMP_SCHEDULE` specifies the type of iteration scheduling to use for DO and PARALLEL DO loop directives that include the SCHEDULE(RUNTIME) clause, described in "SCHEDULE," on page 164. The default value for this variable is STATIC.

If the optional chunk size is not set, a chunk size of 1 is assumed except in the case of a static schedule. For a static schedule, the default is as defined in "DO...END DO ," on page 168.

Examples of the use of OMP_SCHEDULE are as follows:

```
% setenv OMP_SCHEDULE "STATIC, 5"
% setenv OMP_SCHEDULE "GUIDED, 8"
% setenv OMP_SCHEDULE "DYNAMIC"
```

OMP_STACKSIZE

OMP_STACKSIZE is an OpenMP 3.0 feature that controls the size of the stack for newly-created threads. This variable overrides the default stack size for a newly created thread. The value is a decimal integer followed by an optional letter B, K, M, or G, to specify bytes, kilobytes, megabytes, and gigabytes, respectively. If no letter is used, the default is kilobytes. There is no space between the value and the letter; for example, one megabyte is specified 1M. The following example specifies a stack size of 8 megabytes.

```
% setenv OMP_STACKSIZE 8M
```

The API functions related to OMP_STACKSIZE are omp_set_stack_size and omp_get_stack_size.

The environment variable OMP_STACKSIZE is read on program start-up. If the program changes its own environment, the variable is not re-checked.

This environment variable takes precedence over MPSTKZ, which increases the size of the stacks used by threads executing in parallel regions. Once a thread is created, its stack size cannot be changed.

In the PGI implementation, threads are created prior to the first parallel region and persist for the life of the program. The stack size of the main thread (thread 0) is set at program start-up and is not affected by OMP_STACKSIZE.

For more information on controlling the program stack size in Linux, refer to"Running Parallel Program on Linux" in Chapter 2 of the PGI User's Guide. For more information on MPSTKZ, refer to the PGI User's Guide.

OMP_THREAD_LIMIT

You can use the OMP_THREAD_LIMIT environment variable to specify the absolute maximum number of threads that can be used in a parallel program. Attempts to dynamically set the number of processes or threads to a higher value, for example using set_omp_num_threads(), cause the number of processes or threads to be set at the value of OMP_THREAD_LIMIT rather than the value specified in the function call.

OMP_WAIT_POLICY

OMP_WAIT_POLICY sets the behavior of idle threads - specifically, whether they spin or sleep when idle. The values are ACTIVE and PASSIVE, with ACTIVE the default. The behavior defined by OMP_WAIT_POLICY is also shared by threads created by auto-parallelization.

- Threads are considered idle when waiting at a barrier, when waiting to enter a critical region, or when unemployed between parallel regions.
- Threads waiting for critical sections always busy wait (ACTIVE).

- Barriers always busy wait (ACTIVE), with calls to `sched_yield` determined by the environment variable `MP_SPIN`, that specifies the number of times it checks the semaphore before calling `sched_yield()` (on Linux or MAC OS X) or `_sleep()` (on Windows).
- Unemployed threads during a serial region can either busy wait using the barrier (ACTIVE) or politely wait using a mutex (PASSIVE). This choice is set by `OMP_WAIT_POLICY`, so the default is ACTIVE.

When ACTIVE is set, idle threads consume 100% of their CPU allotment spinning in a busy loop waiting to restart in a parallel region. This mechanism allows for very quick entry into parallel regions, a condition which is good for programs that enter and leave parallel regions frequently.

When PASSIVE is set, idle threads wait on a mutex in the operating system and consume no CPU time until being restarted. Passive idle is best when a program has long periods of serial activity or when the program runs on a multi-user machine or otherwise shares CPU resources.

Chapter 9. 3F Functions and VAX Subroutines

The PGI Fortran compilers support FORTRAN 77 3F functions and VAX/VMS system subroutines and built-in functions.

3F Routines

This section describes the functions and subroutines in the Fortran runtime library which are known as 3F routines on many systems. These routines provide an interface from Fortran programs to the system in the same manner as the C library does for C programs. These functions and subroutines are automatically loaded from PGI's Fortran runtime library if referenced by a Fortran program.

The implementation of many of the routines uses functions which reside in the C library. If a C library does not contain the necessary functions, undefined symbol errors will occur at link-time. For example, if PGI's C library is the C library available on the system, the following 3F routines exist in the Fortran runtime library, but use of these routines will result in errors at link-time:

besj0	besj1	besjn	besy0	besy1	besyn
dbesj0	dbesj1	dbesjn	dbesy0	dbesy1	dbesyn
derf	derfc	erf	erfc	getlog	hostnm
lstat	putenv	symlnk	ttynam		

The routines mclock and times depend on the existence of the C function times().

The routines dtime and etime are only available in a SYSVR4 environment. These routines are not available in all environments simply because there is no standard mechanism to resolve the resolution of the value returned by the times() function.

There are several 3F routines, such as fputc and fgetc, that perform I/O on a logical unit. These routines bypass normal Fortran I/O. If normal Fortran I/O is also performed on a logical unit which appears in any of these routines, the results are unpredictable.

abort

Terminate abruptly and write memory image to core file.

Synopsis

```
subroutine abort()
```

Description

The `abort` function cleans up the I/O buffers and then aborts, producing a core file in the current directory.

access

Determine access mode or existence of a file.

Synopsis

```
integer function access(fil, mode)
character*(*) fil
character*(*) mode
```

Description

The `access` function tests the file, whose name is fil, for accessibility or existence as determined by mode.

The mode argument may include, in any order and in any combination, one or more of:

r
: test for read permission

w
: test for write permission

x
: test for execute permission

(blank)
: test for existence

An error code is returned if either the mode argument is illegal or if the file cannot be accessed in all of the specified modes. Zero is returned if the specified access is successful.

alarm

Execute a subroutine after a specified time.

Synopsis

```
integer function alarm(time, proc)
integer time
external proc
```

Description

This routine establishes subroutine proc to be called after time seconds. If time is 0, the alarm is turned off and no routine will be called. The return value of alarm is the time remaining on the last alarm.

Bessel functions

These functions calculate Bessel functions of the first and second kinds for real and double precision arguments and integer orders.

```
besj0
besj1
besjn
besy0
besy1
besyn
dbesj0
dbesj1
dbesjn
dbesy0
dbesy1
dbesyn
```

Synopsis

```
real function besj0(x)
real x
real function besj1(x)
real x
```

```
real function besjn(n, x)
integer n
real x
real function besy0(x)
real x
real function besy1(x)
real x
real function besyn(n, x)
integer n
real x
double precision function dbesj0(x)
double precision x
double precision function dbesj1(x)
double precision x
double precision function dbesjn(n, x)
integer n
double precision x
double precision function dbesy0(x)
double precision x
double precision function dbesy1(x)
double precision x
double precision function dbesyn(n, x)
integer n
double precision x
```

chdir

Change default directory.

Synopsis

```
integer function chdir(path)
character*(*) path
```

Description

Change the default directory for creating and locating files to path. Zero is returned if successful; otherwise, an error code is returned.

chmod

Change mode of a file.

Synopsis

```
integer function chmod(nam, mode)
character*(*) nam
integer mode
```

Description

Change the file system mode of file `nam`. If successful, a value of 0 is returned; otherwise, an error code is returned.

ctime

Return the system time.

Synopsis

```
character*(*) function ctime(stime)
integer*8 stime
```

Description

ctime converts a system time in stime to its ASCII form and returns the converted form. Neither newline nor NULL is included.

date

Return the date.

Synopsis

```
character*(*) function date(buf)
```

Description

Returns the ASCII representation of the current date. The form returned is dd-mmm-yy.

error functions

The functions erf and derf return the error function of x. erfc and derfc return 1.0-erf(x) and 1.0-derf(x), respectively.

Synopsis

```
real function erf(x)
real x
real function erfc(x)
real x
double precision function derf(x)
double precision x
double precision function derfc(x)
double precision x
```

etime, dtime

Get the elapsed time.

Synopsis

```
real function etime(tarray)
real function dtime(tarray)
real tarray(2)
```

Description

etime returns the total processor runtime in seconds for the calling process.

dtime (delta time) returns the processor time since the previous call to dtime. The first time it is called, it returns the processor time since the start of execution.

Both functions place values in the argument tarray: user time in the first element and system time in the second element. The return value is the sum of these two times.

exit

Terminate program with status.

Synopsis

```
subroutine exit(s)
integer s
```

Description

exit flushes and closes all of the program's files, and returns the value of s to the parent process.

fdate

Return date and time in ASCII form.

Synopsis

```
character*(*) function fdate()
```

Description

fdate returns the current date and time as a character string. Neither newline nor NULL will be included.

fgetc

Get character from a logical unit.

Synopsis

```
integer function fgetc(lu, ch)
integer lu
character*(*) ch
```

Description

Returns the next character in ch from the file connected to the logical unit lu, bypassing normal Fortran I/O statements. If successful, the return value is zero; -1 indicates that an end-of-file was detected. Any other value is an error code.

flush

Flush a logical unit.

Synopsis

```
subroutine flush(lu)
integer lu
```

Description

flush flushes the contents of the buffer associated with logical unit lu.

fork

Fork a process.

Synopsis

```
integer function fork()
```

Description

fork creates a copy of the calling process. The value returned to the parent process will be the process id of the copy. The value returned to the child process (the copy) will be zero. If the returned value is negative, an error occurred and the value is the negation of the system error code.

fputc

Write a character to a logical unit.

Synopsis

```
integer function fputc(lu, ch)
integer lu
character*(*) ch
```

Description

A character ch is written to the file connected to logical unit lu bypassing normal Fortran I/O. If successful, a value of zero is returned; otherwise, an error code is returned.

free

Free memory.

Synopsis

```
subroutine free(p)
int p
```

Description

Free a pointer to a block of memory located by malloc; the value of the argument, p, is the pointer to the block of memory.

fseek

Position file at offset.

Synopsis

```
integer function fseek(lu, offset, from)
integer lu
integer offset
integer from
```

Description

fseek repositions a file connected to logical unit lu. offset is an offset in bytes relative to the position specified by from :

0
: beginning of the file

1
: current position

2
: end of the file

If successful, the value returned by fseek will be zero; otherwise, it's a system error code.

ftell

Determine file position.

Synopsis

```
integer function ftell(lu)
integer lu
```

Description

ftell returns the current position of the file connected to the logical unit lu. The value returned is an offset, in units of bytes, from the beginning of the file. If the value returned is negative, it is the negation of the system error code.

gerror

Return system error message.

Synopsis

```
character*(*) function gerror()
```

Description

Return the system error message of the last detected system error.

getarg

Get the nth command line argument.

Synopsis

```
subroutine getarg(n, arg)
integer n
character*(*) arg
```

Description

Return the nth command line argument in arg, where the 0th argument is the command name.

iargc

Index of the last command line argument.

Synopsis

```
integer function iargc()
```

Description

Return the index of the last command line argument, which is also the number of arguments after the command name.

getc

Get character from unit 5.

Synopsis

```
integer function getc(ch)
character*(*) ch
```

Description

Returns the next character in ch from the file connected to the logical unit 5, bypassing normal Fortran I/O statements. If successful, the return value is zero; -1 indicates that an end-of-file was detected. Any other value is an error code.

getcwd

Get pathname of current working directory.

Synopsis

```
integer function getcwd(dir)
character*(*) dir
```

Description

The pathname of the current working directory is returned in dir. If successful, the return value is zero; otherwise, an error code is returned.

getenv

Get value of environment variable.

Synopsis

```
subroutine getenv(en, ev)
character*(*) en
character*(*) ev
```

Description

getenv checks for the existence of the environment variable en. If it does not exist or if its value is not present, ev is filled with blanks. Otherwise, the string value of en is returned in ev.

getgid

Get group id.

Synopsis

```
integer function getgid()
```

Description

Return the group id of the user of the process.

getlog

Get user's login name.

Synopsis

```
character*(*) function getlog()
```

Description

getlog returns the user's login name or blanks if the process is running detached from a terminal.

getpid

Get process id.

Synopsis

```
integer function getpid()
```

Description

Return the process id of the current process.

getuid

Get user id.

Synopsis

```
integer function getuid()
```

Description

Return the user id of the user of the process.

gmtime

Return system time.

Synopsis

```
subroutine gmtime(stime, tarray)
```

```
integer stime
integer tarray(9)
```

Description

Dissect the UNIX time, stime , into month, day, etc., for GMT and return in tarray.

hostnm

Get name of current host.

Synopsis

```
integer function hostnm(nm)
character*(*) nm
```

Description

hostnm returns the name of the current host in nm. If successful, a value of zero is returned; otherwise an error occurred.

idate

Return date in numerical form.

Synopsis

```
subroutine idate(im, id, iy)
integer im, id, iy
```

Description

Returns the current date in the variables im, id, and iy, which indicate the month, day, and year, respectively. The month is in the range 1-12; only the last 2 digits of the year are returned.

ierrno

Get error number.

Synopsis

```
integer function ierrno()
```

Description

Return the number of the last detected system error.

ioinit

Initialize I/O

Synopsis

```
subroutine ioinit(cctl, bzro, apnd, prefix, vrbose)
integer cctl
integer bzro
integer apnd
character*(*) prefix
integer vrbose
```

Description

Currently, no action is performed.

isatty

Is logical unit a tty.

Synopsis

```
logical function isatty(lu)
integer lu
```

Description

Returns *.TRUE.* if logical unit lu is connected to a terminal; otherwise, *.FALSE.* is returned.

itime

Return time in numerical form.

Synopsis

```
subroutine itime(iarray)
integer iarray(3)
```

Description

Return current time in the array iarray. The order is hour, minute, and second.

kill

Send signal to a process.

Synopsis

```
integer function kill(pid, sig)
integer pid
integer sig
```

Description

Send signal number sig to the process whose process id is pid. If successful, the value zero is returned; otherwise, an error code is returned.

link

Make link

Synopsis

```
integer function link(n1, n2)
character*(*) n1
character*(*) n2
```

Description

Create a link n2 to an existing file n1. If successful, zero is returned; otherwise, an error code is returned.

lnblnk

Return index of last non-blank.

Synopsis

```
integer function lnblnk(a1)
character*(*) a1
```

Description

Return the index of the last non-blank character in string a1.

loc

Address of an object.

Synopsis

```
integer function loc(a)
integer a
```

Description

Return the value which is the address of a.

ltime

Return system time.

Synopsis

```
subroutine ltime(stime, tarray)
integer stime
integer tarray(9)
```

Description

Dissect the UNIX time, stime , into month, day, etc., for the local time zone and return in tarray.

malloc

Allocate memory.

Synopsis

```
integer function malloc(n)
integer n
```

Description

Allocate a block of n bytes of memory and return the pointer to the block of memory.

mclock

Get elapsed time.

Synopsis

```
integer function mclock()
```

Description

mclock returns the sum of the user's cpu time and the user and system times of all child processes. The return value is in units of clock ticks per second.

mvbits

Move bits.

Synopsis

```
subroutine mvbits(src, pos, len, dest, posd)
integer src
integer pos
integer len
integer dest
integer posd
```

Description

len bits are moved beginning at position pos of argument src to position posd of argument dest.

outstr

Print a character string.

Synopsis

```
integer function outstr(ch)
```

```
character*(*) ch
```

Description

Output the character string to logical unit 6 bypassing normal Fortran I/O. If successful, a value of zero is returned; otherwise, an error occurred.

perror

Print error message.

Synopsis

```
subroutine perror(str)
character*(*) str
```

Description

Write the message indicated by str to logical unit 0 and the message for the last detected system error.

putc

Write a character to logical unit 6.

Synopsis

```
integer function putc(ch)
character*(*) ch
```

Description

A character ch is written to the file connected to logical unit 6 bypassing normal Fortran I/O. If successful, a value of zero is returned; otherwise, an error code is returned.

putenv

Change or add environment variable.

Synopsis

```
integer function putenv(str)
character*(*) str
```

Description

str contains a character string of the form name=value. This function makes the value of the environment variable name equal to value. If successful, zero is returned.

qsort

Quick sort.

Synopsis

```
subroutine qsort(array, len, isize, compar)
dimension array(*)
integer len
integer isize
external compar
integer compar
```

Description

qsort sorts the elements of the one dimensional array, array. len is the number of elements in the array and isize is the size of an element. compar is the name of an integer function that determines the sorting order. This function is called with 2 arguments (arg1 and arg2) which are elements of array. The function returns:

negative
: if arg1 is considered to precede arg2

zero
: if arg1 is equivalent to arg2

positive
: if arg1 is considered to follow arg2

rand, irand, srand

Random number generator.

Synopsis

```
double precision function rand()
integer function irand()
subroutine srand(iseed)
integer iseed
```

Description

The functions rand and irand generate successive pseudo-random integers or double precision numbers. srand uses its argument, iseed, to re-initialize the seed for successive invocations of rand and irand.

irand
: returns a positive integer in the range 0 through 2147483647.

rand
: returns a value in the range 0 through 1.0.

random, irandm, drandm

Return the next random number value.

Synopsis

```
real function random(flag)
integer flag
integer function irandm(flag)
integer flag
double precision function drandm(flag)
integer flag
```

Description

If the argument, `flag`, is nonzero, the random number generator is restarted before the next random number is generated.

Integer values range from 0 through 2147483647.

Floating point values range from 0.0 through 1.0.

range

Range functions.

Synopsis

```
real function flmin()
real function flmax()
real function ffrac()
double precision function dflmin()
double precision function dflmax()
double precision function dffrac()
integer function inmax()
```

Description

The following range functions return a value from a range of values.

flmin
: minimum single precision value

flmax
: maximum single precision value

ffrac
: smallest positive single precision value

dflmin
: minimum double precision value

dflmax
: maximum double precision value

dffrac
: smallest positive double precision value

inmax
: maximum integer

rename

Rename a file.

Synopsis

```
integer function rename(from, to)
character*(*) from
character*(*) to
```

Description

Renames the existing file from where the new name is, the `from` value, to what you want it to be, the `to` value.. If the rename is successful, zero is returned; otherwise, the return value is an error code.

rindex

Return index of substring.

Synopsis

```
integer function rindex(a1, a2)
character*(*) a1
character*(*) a2
```

Description

Return the index of the last occurrence of string a2 in string a1.

secnds, dsecnds

Return elapsed time.

Synopsis

```
real function secnds(x)
real x
double precision function dsecnds(x)
double precision x
```

Description

Returns the elapsed time, in seconds, since midnight, minus the value of x.

setvbuf

Change I/O buffering behavior.

Synopsis

```
interface
 function setvbuf(lu, typ, size, buf)
 integer setvbuf, lu, typ, size
 character* (*) buf
 end function
 end interface
```

Description

Fortran I/O supports 3 types of buffering:

- Fully buffered: on output, data is written once the buffer is full. On input, the buffer is filled when an input operation is requested and the buffer is empty.
- Line buffered: on output, data is written when a newline character is inserted in the buffer or when the buffer is full. On input, if an input operation is encountered and the buffer is empty, the buffer is filled until a newline character is encountered.
- Unbuffered: No buffer is used. Each I/O operation is completed as soon as possible. In this case, the typ and size arguments are ignored.

Logical units 5 (stdin) and 6 (stdout) are line buffered. Logical unit 0 (stderr) is unbuffered. Disk files are fully buffered. These defaults generally give the expected behavior. You can use setvbuf3f to change a unit's buffering type and size of the buffer.

Note

The underlying stdio implementation may silently restrict your choice of buffer size.

This function must be called after the unit is opened and before any I/O is done on the unit.

The typ parameter can have the following values, 0 specifies full buffering, 1 specifies line buffering, and 2 specifies unbuffered. The size parameter specifies the size of the buffer. Note, the underlying stdio implementation may silently restrict your choice of buffer size.

The buf parameter is the address of the new buffer.

Note

The buffer specified by the buf and size parameters must remain available to the Fortran runtime until after the logical unit is closed.

This function returns zero on success and non-zero on failure.

An example of a program in which this function might be useful is a long-running program that periodically writes a small amount of data to a log file. If the log file is line buffered, you could check the log file for progress. If the log file is fully buffered (the default), the data may not be written to disk until the program terminates.

setvbuf3f

Change I/O buffering behavior.

Synopsis

```
interface
 function setvbuf3f(lu, typ, size)
 integer setvbuf3f, lu, typ, size
 end function
 end interface
```

Description

Fortran I/O supports 3 types of buffering., described in detail in the description of "setvbuf," on page 206 Logical units 5 (stdin) and 6 (stdout) are line buffered. Logical unit 0 (stderr) is unbuffered. Disk files are fully buffered. These defaults generally give the expected behavior. You can use setvbuf3f to change a unit's buffering type and size of the buffer.

Note

The underlying stdio implementation may silently restrict your choice of buffer size.

This function must be called after the unit is opened and before any I/O is done on the unit.

The typ parameter can have the following values, 0 specifies full buffering, 1 specifies line buffering, and 2 specifies unbuffered. The size parameter specifies the size of the buffer.

This function returns zero on success and non-zero on failure.

An example of a program in which this function might be useful is a long-running program that periodically writes a small amount of data to a log file. If the log file is line buffered, you could check the log file for progress. If the log file is fully buffered (the default), the data may not be written to disk until the program terminates.

signal

Signal facility.

Synopsis

```
integer function signal(signum, proc, flag)
integer signum
external proc
integer flag
```

Description

signal allows the calling process to choose how the receipt of a specific signal is handled; signum is the signal and proc is the choice. If flag is negative, proc is a Fortran subprogram and is established as the signal handler for the signal. Otherwise, proc is ignored and the value of flag is passed to the system as the signal action definition. In particular, this is how previously saved signal actions can be restored. There are two special cases of flag: 0 means use the default action and 1 means ignore this signal.

The return value is the previous action. If this is a value greater than one, then it is the address of a routine that was to have been called. The return value can be used in subsequent calls to signal to restore a previous action. A negative return value indicates a system error.

sleep

Suspend execution for a period of time.

Synopsis

```
subroutine sleep(itime)
integer itime
```

Description

Suspends the process for t seconds.

stat, lstat, fstat, fstat64

Get file status.

Synopsis

```
integer function stat(nm, statb)
character*(*) nm
integer statb(*)

integer function lstat(nm, statb)
character*(*) nm
integer statb(*)

integer function fstat(lu, statb)
integer lu
integer statb(*)

integer function fstat64(lu, statb)
integer lu
integer*8 statb(*)
```

Description

Return the file status of the file in the array statb. If successful, zero is returned; otherwise, the value of -1 is returned. stat obtains information about the file whose name is nm; if the file is a symbolic link, information is obtained about the file the link references. lstat is similar to stat except lstat returns information about the link. fstat obtains information about the file which is connected to logical unit lu.

stime

Set time.

Synopsis

```
integer function stime(tp)
integer tp
```

Description

Set the system time and date. tp is the value of the time measured in seconds from 00:00:00 GMT January 1, 1970.

symlnk

Make symbolic link.

Synopsis

```
integer function symlnk(n1, n2)
character*(*) n1
character*(*) n2
```

Description

Create a symbolic link n2 to an existing file n1. If successful, zero is returned; otherwise, an error code is returned.

system

Issue a shell command.

Synopsis

```
integer function system(str)
character*(*) str
```

Description

system causes the string, str, to be given to the shell as input. The current process waits until the shell has completed and returns the exit status of the shell.

time

Return system time.

Synopsis

```
integer*8 function time()
```

Description

Return the time since 00:00:00 GMT, January 1, 1970, measured in seconds.

times

Get process and child process time

Synopsis

```
integer function times(buff)
integer buff(*)
```

Description

Returns the time-accounting information for the current process and for any terminated child processes of the current process in the array buff. If successful, zero is returned; otherwise, the negation of the error code is returned.

ttynam

Get name of a terminal

Synopsis

```
character*(*) ttynam(lu)
integer lu
```

Description

Returns a blank padded path name of the terminal device connected to the logical unit lu. The lu is not connected to a terminal, blanks are returned.

unlink

Remove a file.

Synopsis

```
integer function unlink(fil)
character*(*) fil
```

Description

Removes the file specified by the pathname fil. If successful, zero is returned; otherwise, an error code is returned.

wait

Wait for process to terminate.

Synopsis

```
integer function wait(st)
integer st
```

Description

wait causes its caller to be suspended until a signal is received or one of its child processes terminates. If any child has terminated since the last wait, return is immediate. If there are no child processes, return is immediate with an error code.

If the return value is positive, it is the process id of the child and st is its termination status. If the return value is negative, it is the negation of an error code.

VAX System Subroutines

The PGI FORTRAN77 compiler, pgf77, supports a variety of built-in functions amd VAX/VMS system subroutines.

Built-In Functions

The built-in functions perform inter-language utilities for argument passing and location calculations. The following built-in functions are available:

%LOC(arg)

Compute the address of the argument arg.

%REF(a)

Pass the argument a by reference.

%VAL(a)

Pass the argument as a 32-bit immediate value (64-bit if a is double precision.) A value of 64-bits is also possible if supported for integer and logical values.

VAX/VMS System Subroutines

DATE

The DATE subroutine returns a nine-byte string containing the ASCII representation of the current date. It has the form:

```
CALL DATE(buf)
```

where buf is a nine-byte variable, array, array element, or character substring. The date is returned as a nine-byte ASCII character string of the form:

```
dd-mmm-yy
```

Where:

dd
: is the two-digit day of the month

mmm
: is the three-character abbreviation of the month

yy
is the last two digits of the year

EXIT

The EXIT subroutine causes program termination, closes all open files, and returns control to the operating system. It has the form:

```
CALL EXIT[(exit_status)]
```

where:

exit_status
is an optional integer argument used to specify the image exit value

GETARG

The GETARG subroutine returns the Nth command line argument in character variable ARG. For N equal to zero, the name of the program is returned.

```
SUBROUTINE GETARG(N, ARG)
INTEGER*4 N
CHARACTER*(*) ARG
```

IARGC

The IARGC subroutine returns the number of command line arguments following the program name.

```
INTEGER*4 FUNCTION IARGC()
```

IDATE

The IDATE subroutine returns three integer values representing the current month, day, and year. It has the form:

```
CALL IDATE(IMONTH, IDAY, IYEAR)
```

If the current date were October 9, 2004, the values of the integer variables upon return would be:

```
IMONTH = 10
IDAY = 9
IYEAR = 04
```

MVBITS

The MVBITS subroutine transfers a bit field from one storage location (source) to a field in a second storage location (destination). MVBITS transfers a3 bits from positions a2 through (a2 + a3 - 1) of the source, src, to positions a5 through (a5 + a3 - 1) of the destination, dest. Other bits of the destination location remain unchanged. The values of (a2 + a3) and (a5 + a3) must be less than or equal to 32 (less than or equal to 64 if the source or destination is INTEGER*8). It has the form:

```
CALL MVBITS(src, a2, a3, dest, a5)
```

Where:

src
is an integer variable or array element that represents the source location.

a2
: is an integer expression that identifies the first position in the field transferred from *src*.

a3
: is an integer expression that identifies the length of the field transferred from *src*.

dest
: is an integer variable or array element that represents the destination location.

a5
: is an integer expression that identifies the starting position within *a4*, for the bits being transferred.

RAN

The RAN subroutine returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1. The result is a floating point number that is uniformly distributed in the range between 0.0 and 1.0 exclusive. It has the form:

```
y = RAN(i)
```

where y is set equal to the value associated by the function with the seed argument i. The argument i must be an INTEGER*4 variable or INTEGER*4 array element.

The argument i should initially be set to a large, odd integer value. The RAN function stores a value in the argument that it later uses to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs in order to obtain different random numbers. The seed is updated automatically, and RAN uses the following algorithm to update the seed passed as the parameter:

```
SEED = 6969 * SEED + 1 ! MOD
2**32
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

If the command-line option to treat all REAL declarations as DOUBLE PRECISION declarations is in effect, RAN returns a DOUBLE PRECISION value.

SECNDS

The SECNDS subroutine provides system time of day, or elapsed time, as a floating point value in seconds. It has the form:

```
y = SECNDS(x)
```

where (REAL or DOUBLE PRECISION) y is set equal to the time in seconds since midnight, minus the user supplied value of the (REAL or DOUBLE PRECISION) x. Elapsed time computations can be performed with the following sequence of calls.

```
X = SECNDS(0.0)
...
... ! Code to be timed
...
DELTA = SECNDS(X)
```

The accuracy of this call is the same as the resolution of the system clock.

TIME

The TIME subroutine returns the current system time as an ASCII string. It has the form:

```
CALL TIME(buf)
```

where buf is an eight-byte variable, array, array element, or character substring. The TIME call returns the time as an eight-byte ASCII character string of the form:

```
hh:mm:ss
```

For example:

```
16:45:23
```

Note that a 24-hour clock is used.

Chapter 10. Interoperability with C

Fortran 2003 provides a mechanism for interoperating with C. Any entity involved must have equivalent declarations made in both C and Fortran. This section describes the pointer types and enumerators available for interoperability.

Enumerators

Fortran 2003 has enumerators to provide interoperability with C. An enumerator is a set of integer constants. The kind of enumerator corresponds to the integer type that C would choose for the same set of constants.

You can specify the value of an enumerator when it is declared. If the value is not specified, then it is one of two values:

- If the enumerator is the first one, the value is zero.
- If the enumerator is not the first one, the value is one greater than the previous enumerator.

Example 10.1. Enumerator Example

In the following example, green has a value of 4, purple a value of 8, and gold a value of 9.

```
enum, bind(c)
   enumerator :: green = 4, purple = 8
   enumerator gold
end enu
```

Interoperability with C Pointer Types

C pointers are addresses. The derived type `c_ptr` is interoperable with C pointer types. The named constant `c_null_ptr` corresponds to the null value in C.

c_f_pointer

A subroutine that assigns the C pointer target to the Fortran pointer, and specifies its shape.

F2003

Syntax

```
c_f_pointer (cptr, fptr [,shape] )
```

Type

subroutine

Description

c_f_pointer assigns the target, cptr, to the Fortran pointer, fptr, and specifies its shape.

- `cptr` is a scalar of the type C_PTR with INTENT(IN). Its value is one of the following:
  - the C address of an interoperable data entity.

    Note

    If cptr is the C address of a Fortran variable, the Fortran variable must have the `target` attribute.
  - the result of a reference to `c_loc` with a non-interoperable argument.
- `ftpr` is a pointer with INTENT(OUT).
  - If `cptr` is the C address of an interoperable data entity, then `fptr` must be a data pointer of the type and type parameters of the entity. It becomes pointer associated with the target of `cptr`.
  - If `cptr` was returned by a call of `c_loc` with a non-interoperable argument `x`, then `fptr` must be a nonpolymorphic scalar pointer of the type and type parameters of `x`.

    Note

    `x` or its target if it is a pointer, must not have been deallocated or become undefined due to execution of a `return` or `end` statement.

    `fptr` is associated with x or its target.
- `shape` is an optional argument that is a rank-one array of type INTEGER with INTENT(IN). `Shape` is present if and only if `fptr` is an array. The size must be equal to the rank of `fptr`; each lower bound is assumed to be 1.

Example

```
program main
   use iso_c_binding
   implicit none
   interface
   subroutine my_routine(p) bind(c,name='myC_func')
     import :: c_ptr
     type(c_ptr), intent(out) :: p
   end subroutine
   end interface
     type(c_ptr) :: cptr
     real,pointer :: a(:)
     call my_routine(cptr)
     call c_f_pointer(cptr, a, [12])
    end program main
```

c_f_procpointer

A subroutine that associates a procedure pointer with the target of a C function pointer.

F2003

Syntax

```
c_f_procpointer (cptr, fptr )
```

Type

subroutine

Description

c_f_procpointer associates a procedure pointer with the target of a C function pointer.

- `cptr` is a scalar of the type C_PTR with INTENT(IN). Its value is the C address of the procedure that is interoperable.

  Its value is one of the following:

  - the C address of an interoperable procedure.
  - the result of a reference to `c_loc` with a non-interoperable argument. In this case, there is no intent that any use of it be made within C except to pass it back to Fortran, where C_F_POINTER is available to reconstruct the Fortran pointer.
- `ftpr` is a procedure pointer with INTENT(OUT).

  The interface for `ftpr` shall be interoperable with the prototype that describes the target of `cptr`.

  `ftpr` becomes pointer associated with the target of `cptr`

  - If `cptr` is the C address of an interoperable procedure, then the interface for `ftpr` shall be interoperable with the prototype that describes the target of `cptr`. `fptr` must be a data pointer of the type and type parameters of the entity. It becomes pointer associated with the target of `cptr`.
  - If `cptr` was returned by a call of `c_loc` with a non-interoperable argument `x`, then `fptr` must be a nonpolymorphic scalar pointer of the type and type parameters of `x`.

    #### Note

    `x` or its target if it is a pointer, must not have been deallocated or become undefined due to execution of a `return` or `end` statement.

    `fptr` is associated with x or its target.

Example

```
program main
   use iso_c_binding
   implicit none
   interface
```

```
    subroutine my_routine(p) bind(c,name='myC_func')
      import :: c_ptr
      type(c_ptr), intent(out) :: p
    end subroutine
    end interface
      type(c_ptr) :: cptr
      real,pointer :: a(:)
      call my_routine(cptr)
      call c_f_pointer(cptr, a, [12])
     end program main
```

c_associated

A subroutine that determines the status of a C_PTR, or determines if one C_PTR is associated with a target C_PTR.

F2003

Syntax

```
c_associated (cptr1[, cptr2] )
```

Type

subroutine

Description

c_associated determines the status of a C_PTR,`cptr1`, or determines if `cptr1` is associated with a target `cptr2`.

- `cptr1` is a scalar of the type C_PTR.
- `cptr2` is an optional scalar or the same type as `cptr1`.

Return Value

A logical value:

- .false. if either `cptr1` is a C NULL pointer or if `cptr1` and `cptr2` point to different addresses.
- .true. if `cptr1` is a not a C NULL pointer or if `cptr1` and `cptr2` point to the same address.

Example

```
program main
    use iso_c_binding
    subroutine test_association(h,k)
       only: c_associated, c_loc, c_ptr
       real, pointer :: h
       type(c_ptr) :: k
       if(c_associated(k, c_loc(h))) &
           stop 'h and k do not point to same target'
     end subroutine test_association
```

Interoperability of Derived Types

For a derived type to be interoperable, the following must be true:

- It must have the `bind` attribute.

```
type, bind(c) :: atype
    :
end type atype
```

- It cannot be a sequence type.
- It cannot have type parameters.
- It cannot have the `extends` attribute.
- It cannot have any type-bound procedures.
- Each component must comply with these rules:
  - Must have interoperable type and type parameters.
  - Must not be a pointer.
  - Must not be allocatable.

Under the preceding conditions the type can interoperate with a C struct type that has the same number of components, with components corresponding by their position in the definitions. Further, each Fortran component must be interoperable with its corresponding C component. The name of the type and names of the components is not significant for interoperability.

There is no Fortran type that is interoperable with these C types:

- a C union type,
- a C struct type that contains a bit field
- a C struct type that contains a flexible array member.

Example 10.2. Derived Type Interoperability

This type...	Is interoperable with this type

`typedef struct {`
`  int a,b;`
`  float t;`
`} my_c_type`
	`use iso_c_binding`
	`type, bind(c) :: my_fort_type`
	`  integer(c_int) :: i,j`
	`  real(c_float) :: s`
	`end type my_fort_type`

Index

Symbols

A

B

C

D

E

F

H

I

L

M

N

O

NOTICE

TRADEMARKS

COPYRIGHT