SPARCompiler Pascal 3.0.2

Reference Manual

SunPro
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 801-5055-10
Revision A, December 1993

2550 Garcia Avenue, Mountain View, California 94043-1100

The OPEN LOOK and the Sun Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 (October 1988) and FAR 52.227-19 (June 1987).

The product described in this manual may be protected by one or more U.S. patents, foreign patents, and/or pending applications.

TRADEMARKS

The Sun logo, Sun Microsystems, Sun Workstation, NeWS, and SunLink are registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Sun, Sun-2, Sun-3, Sun-4, Sun386*i*, SunCD, SunInstall, SunOS, SunPro, SunView, NFS, and OpenWindows are trademarks of Sun Microsystems, Inc.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

PostScript is a registered trademark of Adobe Systems Incorporated. Adobe also owns copyrights related to the PostScript language and the PostScript interpreter. The trademark PostScript is used herein only to refer to material supplied by Adobe or to programs written in the PostScript language as defined by Adobe.

X Window System is a product of the Massachusetts Institute of Technology.

SPARC is a registered trademark of SPARC International, Inc. Products bearing the SPARC trademark are based on an architecture developed by Sun Microsystems, Inc. SPARCstation is a trademark of SPARC International, Inc., licensed exclusively to Sun Microsystems, Inc.

All other products or services mentioned in this document are identified by the trademarks, service marks, or product names as designated by the companies who market those products. Inquiries concerning such trademarks should be made directly to those companies.

Contents

Figures

Tables

Preface

Pascal[1] is an implementation of the Pascal language that includes all the standard language elements and many extensions. These extensions allow greater flexibility in writing programs. Among these extensions are the following:

- Separate compilation of programs and modules
- `dbx` (symbolic debugger) support (which is part of SPARCworks)
- Optimizer support
- Multiple `label`, `const`, `type`, and `var` declarations
- Variable-length character strings
- Compile-time initializations
- `static` and `extern` declarations
- Additional sizes of integer and real data types
- Integers in any base from 2 to 16
- Extended input/output facilities
- Extended library of built-in functions and procedures
- Universal and function and procedure pointer types

1. All references to Pascal in this manual will refer to SPARCompiler Pascal unless otherwise indicated.

- Specification of the direction of parameter passing as one of these:
  - into a routine
  - out of a routine
  - both of the above

In addition, Pascal provides a compiler switch to provide compatibility with Apollo Pascal to ease the task of porting your Apollo Pascal applications to workstations.

Who Needs This Manual?

The purpose of this manual is to help programmers understand Pascal. Before you attempt to use this manual, you should be familiar with ISO standard Pascal and SunOS commands and concepts. This manual describes the SPARCompiler Pascal product running in the Solaris environment. SPARCompiler Pascal runs on the Solaris 2.2 environment or later.

Operating Environment

Solaris 2.2 implies:

- SunOS 5.2™ operating system
- A SPARC™ computer, either a server or a workstation
- The OpenWindows™ 3.0 application development platform

SunOS 5.2 is based on the System V Release 4 (SVR4) UNIX[2] operating system, and the ONC™ family of published networking protocols and distributed services. SunOS 4.1.*x* is based on the UCB BSD 4.3 operating system.

Note – For other operating system release-specific information, please refer to the Product Notes and `README` files that accompanies your release of the product.

---

2. UNIX is a registered trademark of UNIX System Laboratories, Inc.

Installation

For instructions on how to install Pascal, and if you are new to the operating system, refer to the documentation on the operating system and on installing the software *Installing SunPro Software on Solaris.*

How This Manual Is Organized

This manual is a reference manual for Pascal extensions to Standard Pascal. Chapters 1 through 7 describe extensions to the elements of a Pascal program:

- Chapter 1: Lexical Elements
- Chapter 2: Data Types
- Chapter 3: Statements
- Chapter 4: Assignments and operators
- Chapter 5: Program Headings and Declarations
- Chapter 6: Built-in Procedures and Functions
- Chapter 7: Input and Output.

As each extension is presented, a small, complete example is provided to illustrate that extension.

The manual also has several appendices and a Glossary:

- Appendix A presents the Pascal Language in meta symbols.
- Appendix B summarizes the Pascal extensions to standard Pascal. This appendix has been included as a quick-reference guide to the differences between Pascal and standard Pascal.
- Appendix C lists the differences between Pascal and Apollo[3] DOMAIN[4] Pascal.
- Appendix D describes Pascal features that are implementation-defined.
- Appendix E lists the incompatibilities between Pascal 1.1 and Pascal 3.0.

3. Apollo is a registered trademark of Apollo Computer Inc., a subsidiary of Hewlett Packard Company.

4. DOMAIN is a registered trademark of Apollo Computer Inc., a subsidiary of Hewlett Packard Company.

Conventions Used in This Manual

This manual contains syntax diagrams of the Pascal language in extended Backus-Naur Formalism (BNF) notation. The manual uses the following meta symbols:

Meta Symbol	Description
::=	Defined as
|	Can be used as an alternative
(*a* | *b*)	Either *a* or b
[*a*]	Zero or one instance of *a*
{ *a* }	Zero or more instances of *a*
'abc'	The keyword abc

The manual also uses the following conventions:

`Bold face typewriter font`

Indicates commands that you should type in exactly as printed in the manual.

`Regular typewriter font`

Represents what the system prints on your workstation screen, as well as the Pascal keywords, identifiers, program names, and filenames.

Italic font

Indicates variables or parameters that you should replace with an appropriate word or string. Also used for emphasis.

`hostname%`

Represents your system prompt for a non-privileged user account.

Screen box

Encloses a program listing or text that represents an interactive session. In the interactive examples, user input is indicated by **`boldface typewriter font`**.

Related Documentation

This manual was designed to accompany the *Pascal 3.0.2 User's Guide*. The *Pascal 3.0.2 User's Guide* describes how to write, compile, run, and debug a Pascal program.

The *Standard Pascal User Reference Manual* (Norton, 1983) by Doug Cooper describes the ISO Pascal Standard and can also be used for reference.

Pascal also provides on-line documentation for the `pc` command. `man pc` describes the Pascal compiler.

The on-line documentation is included in the Pascal package and must be installed with the rest of the software. Once you install the documentation, you can read about `pc` by entering

```
hostname% man pc
```

The following set of manuals contain information about debugging and browsing source code:

SPARCworks Document Set

The following SPARCworks™ manuals contain information on the debuggers, source code browser, and other development tools that are in the SPARCworks package.

SPARCworks Tutorial
Browsing Source Code
Debugging a Program
Building Programs with MakeTool
Managing SPARCworks Tools
Merging Source Files
Performance Tuning an Application

The following manual contains information useful for developing Pascal programs under XView:

XView Programming Manual: An OPEN LOOK Toolkit for X11, by Dan Heller, O'Reilly & Associates, Inc., 1989.

Lexical Elements 1

This chapter describes the symbols and words of a Pascal program.

Character Set

Pascal uses the standard seven-bit ASCII character set, and the compiler distinguishes between upper case and lower case characters. For example, the following seven words are distinct from the predefined type `integer`:

Integer	INTEGer
INteger	INTEGEr
INTeger	INTEGER
INTEger	

If you change the case of characters used in a word, the compiler does not recognize the word and gives an error.

The Pascal keywords and built-in procedure and function names are all lower case.

Two `pc` command-line options allow you to map all keywords and identifiers to lower case when you compile your program:

`-L` Maps all upper case letters in keywords and identifiers to lower case.

`-s` Performs the same action as `-L` and also produces warning diagnostics for nonstandard constructs and extensions.

1

See the *Pascal 3.0.2 User's Guide* for a complete description of pc and its options.

Special Symbols

Pascal recognizes the following standard Pascal symbols and the nonstandard special symbols listed in Table 1-1.

```
+ - * / = < > [ ] . , :=
: ; ( ) <> <= >= .. ^
```

Table 1-1 Nonstandard Special Symbols

Symbol	Description	Example
~	Bitwise not operator.	~ 4
&	Bitwise and operator.	4 & 3
|	Bitwise or operator.	4 | 3
!	Bitwise or operator.	4 ! 3
#	Specifies an integer value in a base other than base 10.	p := 2#10111; { base 2 } f := 8#76543; { base 8 }
	Includes a file in the program.	#include "globals.h" #include "scanner.h"

	Indicates a preprocessor command.	#ifdef DEBUGGING
		writeln('Total :',i,sum);
		#endif

%	Indicates a cppas compiler directive	%var one, two %enable two

Reserved Words

Pascal reserves the standard words in Table 1-2. You may not redefine a reserved word to represent another item.

Table 1-2 Standard Reserved Words

Pascal Standard Reserved Words				
and	else	label	procedure	until
array	file	main	program	var
begin	for	mod	record	while
case	forward	nil	repeat	with
const	function	not	set	
div	goto	of	then	
do	if	or	to	
downto	in	packed	type	

Pascal also reserves the nonstandard words in Table 1-3. These words are not treated as reserved words when you compile your program with any of the `-s`, `-s0`, `-s1`, `-V0` or `-V1` options.

Table 1-3 Nonstandard Reserved Words

Pascal Nonstandard Reserved Words	
define	private
extern	public
external	static
module	univ
otherwise	

Identifiers

In Pascal, you can include a dollar sign ($) and underscore (_) in an identifier name. The $ and _ may occur in any position of the identifier name. However, you should avoid using these characters in the first position because they may conflict with system names.

Pascal predeclares the standard identifiers in Table 1-4 and the nonstandard identifiers in Table 1-5.

Table 1-4 Predeclared Standard Identifiers

Pascal Predeclared Standard Identifiers				
abs	exp	ord	rewrite	write
arctan	false	output	round	writeln
boolean	get	page	sin	
char	input	pred	sqr	
chr	integer	put	sqrt	
cos	ln	read	succ	
dispose	maxint	readln	text	
eof	new	real	true	
eoln	odd	reset	trunc	

Table 1-5 Predeclared Nonstandard Identifiers

Pascal Predeclared Nonstandard Identifiers				
FALSE	exit	longreal	return	xor
TRUE	expo	lor	rshft	
addr	firstof	lshft	seed	
alfa	flush	lsl	shortreal	
argc	getenv	lsr	single	
argv	getfile	max	sizeof	
arshft	halt	maxchar	stlimit	
asl	in_range	message	string	
asr	index	min	substr	
assert	integer16	minchar	sysclock	
bell	integer32	minint	tab	
card	intset	next	time	
clock	land	null	trim	
close	lastof	open	univ_ptr	
date	length	pack	unpack	
discard	linelimit	random	varying	
double	lnot	remove	wallclock	

You may redefine a predeclared identifier to represent another item. For example, you could redefine the predefined identifier `next`, a statement that causes the program to skip to the next iteration of the current loop, as a variable.

Once you redefine an identifier, you cannot use it as originally defined in the program, as shown in the following example:

The Pascal program, `pred_iden.p`, redefines the predeclared identifier `next` as an integer variable.

```
program predefined_identifier;

var
    i: integer;
    next: integer;

begin
    for i := 1 to 10 do begin
      if i > 5 then begin
          next
      end
    end
end. { predefined_identifier }
```

This program does not compile because `next` is declared as a variable but used in its original definition as a statement.

```
hostname% pc pred_iden.p
Fri Feb 14 15:13:17 1992  pred_iden.p:
         10           next
E 18470-----------------^---  Replaced variable id with a
procedure id
In program predefined_identifier:
E 18250 next improperly used on line 10
hostname%
```

Comments

In Pascal, you can specify a comment in either braces, quotation marks, a parenthesis / asterisk pair, or a slash / asterisk pair:

```
{  This is a comment. }
(* This is a comment. *)
"  This is a comment. "
```

```
/* This is a comment. */
```

The symbols used to delimit a comment must match. For example, a comment that starts with `{` must end with `}`, and a comment that starts with `(*` must end with `*)`.

Pascal allows you to nest comments; you can include one type of comment delimiter inside another:

```
{ This is a valid (* comment within a comment. *) }
(* This is a valid " comment within a comment. " *)
```

You may not nest the same kind of comments. The following comments result in a compile-time error:

```
{  This is not a valid { comment within a comment. } }
(* This is not a valid (* comment within a comment. *) *)
"  This is not a valid " comment within a comment. " "
/* This is not a valid /* comment within a comment. */ */
```

Data Types 2

This chapter describes the Pascal data types. You will note that some data types represent different values when you compile your program with and without the `–xl` option and with and without the `–calign` option. The intent of the `–xl` option is to guarantee binary data compatibility between the operating system and Apollo MC680x0-based workstations. The intent of the `-calign` option is to improve compatibility with C language data structures.

Summary of Data Format Differences

A few data formats, particularly of structured types, change when you use the Pascal compiler's `-calign` option, when you use the `-xl` option, and when you use the `-calign` with the `-xl` option. This section describes the data alignments and sizes that change with these options. See the remainder of the chapter for information on types that do not change when you use these options.

All simple data types take their natural alignments. For example, real numbers, being four-byte values, have four-byte alignment. Naturally, no padding is needed for simple types.

2

Default Data Alignments and Padding

Records

The alignment of a record is always four bytes. Elements take their natural alignment, but the total size of a record is always a multiple of four bytes.

Packed Records

Elements of types enumerated, subrange, `integer16`, and sets with a cardinal number less than 32 are bit-aligned in packed records.

Variant Records

The alignment of each variant in a record is the maximum alignment of all variants.

Arrays

The alignment of a array is equal to the alignment of the elements, and the size of most arrays is simply the size of each element times the number of elements. The one exception to this rule is that the arrays of aggregates always have a size that is a multiple of four bytes.

Sets

Sets have an alignment of four bytes. The size of a set is always a multiple of two bytes.

Enumerated Types

The size and alignment of enumerated types can be one byte or two, depending on the number of elements defined for the type.

Subranges

The size and alignment of subrange types varies from one to four bytes, depending on the number of bits requires for its minimum and maximum values. See Table 2-7 for examples.

Data Formats with `-calign`

Records

The alignment of a record is equal to the alignment of the largest element.

Packed Records

Packed records are the same as the default, except integer elements are not bit-aligned.

Arrays

The size of all arrays is the size of each element times the number of elements.

Sets

Sets have an alignment of two bytes. The size is the same as the default.

Data Formats with `-xl`

In addition to the structured types discussed below, two simple data types change their size with the `-xl` option.

Type `real` is eight bytes by default; with `-xl`, it is four bytes.

Type integer is four bytes by default; with `-xl`, it is two bytes.

Packed Records

Values of type `real` have 4 bytes size and alignment. Values of type `integer` have a size of two bytes and are bit-aligned.

Enumerated Types

The size and alignment of enumerated types is always two bytes.

Subranges

The size and alignment of subrange types varies from two to four bytes, depending on the number of bits requires for its minimum and maximum values. See Table 2-7 for examples.

Data Formats with `-calign` *and* `-xl`

When you use `-xl` with `-calign`, alignments and padding are the same as with `-xl` alone, with the differences noted below.

Arrays

Arrays are the same as with `-calign` alone, except the size of an array of booleans is always a multiple of two.

Varying Arrays

Varying arrays have an alignment of four bytes. The size is a multiple of four.

Real

Pascal supports the standard predeclared `real` data type. As extensions to the standard, Pascal also supports the `single`, `shortreal`, `double`, and `longreal` data types, real initialization in the variable declaration, and real constants without a digit after the decimal point.

Real Variables

The minimum and maximum values of the real data types is shown in Table 2-1.

Table 2-1 Real Data Types

Type	Bits	Maximum Value	Minimum Value
real (with –xl option)	32 b-its	3.402823e+38	1.401298e-45
`real` (without `-xl` option)	64 bits	1.79769313486231470e+308	4.94065645841246544e-324
`single`	32 bits	3.402823e+38	1.401298e-45
`shortreal`	32 bits	3.402823e+38	1.401298e-45
`double`	64 bits	1.79769313486231470e+308	4.94065645841246544e-324
`longreal`	64 bits	1.79769313486231470e+308	4.94065645841246544e-324

This example declares five real variables:

```
var x: real;
    y: shortreal;
    z: longreal;
    weight: single;
    volume: double;
```

Real Initialization

To initialize a real variable when you declare it in the `var` declaration of your program, create an assignment statement as follows:

This example initializes the variable ph to 4.5 and y to 2.71828182845905e+00.

```
var
    ph: single := 4.5;
    y: longreal := 2.71828182845905e+00;
```

You can also initialize real variables in the `var` declaration of a procedure or function; however, when you do so, you must also declare the variable as `static`:

This example initializes the variable `sum` to 5.0, which has been declared as `static single`:

```
procedure foo (in x : single;
    out y: single);

 var
    sum: static single := 5.0;
```

The following example defines six valid real constants, two of which do not have a digit after the decimal point.

Real Constants

Here is an example that of a real contstant:

```
const
    n = 42.57;
    n2 = 4E12;
    n3 = 567.;
    n4 = 83.;
    n5 = cos(567.)/2;
          n6 = succ(sqrt(5+4));
```

Data Representation

Pascal represents `real`, `single`, `shortreal`, `double`, and `longreal` data types according to the IEEE standard, *A Standard for Binary Floating-Point Arithmetic.* Figure 2-1 shows the representation of a 32-bit floating point number, and Figure 2-2 shows the representation of a 64-bit floating point number.

Figure 2-1 32-Bit Floating-Point Number

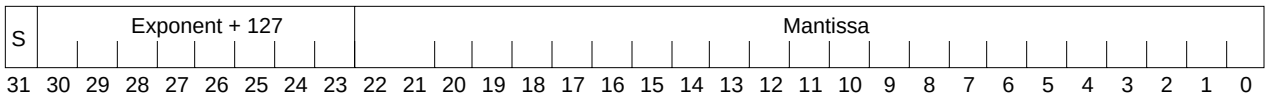
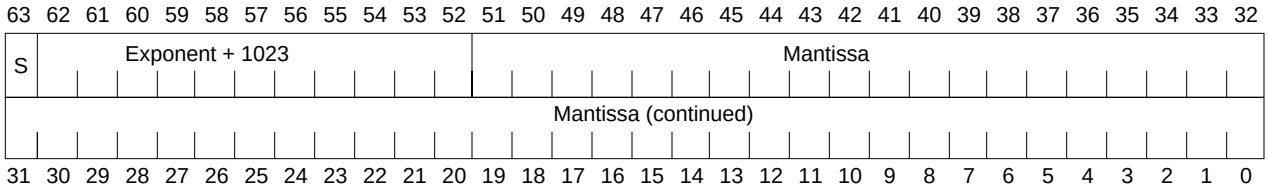


Figure 2-2 64-Bit Floating-Point Number



A real number is represented by this form:

$$(-1)^{sign} * 2^{exponent\text{-}bias} * 1.f$$

f is the bits in the fraction. Extreme exponents are represented as shown in Table 2-2.

Table 2-2 Representation of Extreme Exponents

Exponent	Description
zero (signed)	Represented by an exponent of zero and a fraction of zero.
Subnormal number	Represented by (-1) ^{sign} * 2 $^{1-bias}$ *0.f where *f* is the bits in the significand.
Signed infinity	(That is, affine infinity.) Represented by the largest value that the exponent can assume (all ones), and a zero fraction.
Not a Number (NaN)	Represented by the largest value that the exponent can assume (all ones), and a nonzero fraction.

Normalized real numbers have an implicit leading bit that provides one more bit of precision than usual.

Table 2-3 shows the hexadecimal representation of several numbers.

Table 2-3 Hexadecimal Representation of Selected Numbers

Value	32-bit Floating-Point Number	64-bit Floating-Point Number
+0	00000000	0000000000000000
-0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
-1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000
+Infinity	7F800000	7FF0000000000000
-Infinity	FF800000	FFF0000000000000
NaN	7Fxxxxxx	7FFxxxxxxxxxxxx

Integer

Pascal supports the standard predeclared `integer` data type. As extensions to the standard, Pascal also supports the `integer16` and `integer32` data types, integer initialization in the variable declaration, and integer constants in a base other than base 10.

Integer Variables

Table 2-4 lists the minimum and maximum values of the integer data types.

Table 2-4 Integer Data Types

Type	Number of Bits	Maximum Value	Minimum Value
`integer`(without the –xl option)	32 bits	2147483647	-2147483648
`integer`(with `–xl` option)	16 bits	32767	-32768
`integer16`	16 bits	32767	-32768
`integer32`	32 bits	2147483647	-2147483648

This example declares three integer variables:

```
var
    i: integer;
    score: integer16;
    number: integer32;
```

To define an unsigned integer in Pascal, use a subrange declaration. The subrange syntax indicates the lower and upper limits of the data type as follows:

This code limits the legal values for the variable `unsigned_int` to 0 through 65536:

```
type
    unsigned_int = 0..65536;
var
    u: unsigned_int;
```

Integer Initialization

To initialize integer variables when you declare them in the `var` declaration part of your program, put an assignment statement in the declaration as follows:

This example initializes the variables a and b to 50 and c to 10000.

```
var   a, b: integer32 := 50;
      c: integer16 := 10000;
```

You may also initialize integer variables in the `var` declaration of a procedure or function; however, when you do so, you must also declare the variable as `static`:

This code initializes the variable `sum` to 50, which has been declared as `static integer16`.

```
procedure show (in x : integer16;
       out y: integer16);
 var
    sum: static integer16 := 50;
```

Integer Constants

You define integer constants in Pascal the same as you do as in standard Pascal:

```
const
    x = 10;
    y = 15;
    n1 = sqr(x);
    n2 = trunc((x+y)/2);
    n3 = arshft(8, 1);
```

`maxint` and `minint`

The value Pascal assigns to the integer constants `maxint` and `minint` is shown in Table 2-5.

Table 2-5 Values for `maxint` and `minint`

	Without `-xl`		With `-xl`	
Constant	**Bits**	**Value**	**Bits**	**Value**
`maxint`	32 bits	2147483647	16 bits	3267
`minint`	32 bits	-2147483648	16 bits	-32768

In Another Base

To specify an integer constant in another base, use the following format:

`base#number`

base is an integer from 2 to 36. *number* is a value in that base. To express *number*, use the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and then use the letters *a* to *z*. Case is insignificant; `a` is equivalent to `A`.

You may optionally put a positive sign (`+`) or negative sign (`-`) before the *base*. The sign applies to the entire number, not the base.

This code specifies integers in binary, octal, and hexadecimal notation.

```
power := 2#10111;            (* binary (base 2) *)
fraction_of_c := -8#76543;   (* octal (base 8) *)
percentage := +16#fd9c;      (* hexadecimal (base 16) *)
```

Data Representation

Pascal represents `integer`, `integer16`, and `integer32` data types in twos complement format. Figure 2-3 shows the representation of a 16-bit integer. Similarly, Figure 2-4 shows the representation of a 32-bit integer.

Figure 2-3 16-Bit Integer

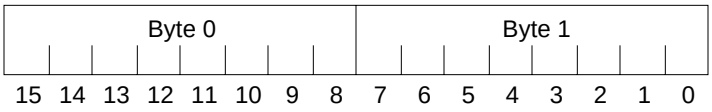
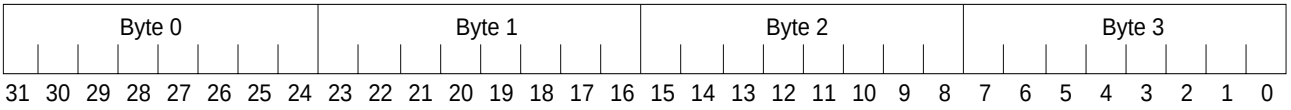


Figure 2-4 32-Bit Integer



Boolean

Pascal supports the standard predeclared data type `boolean`. As an extension to the standard, Pascal permits you to initialize boolean variables in the variable declaration.

Boolean Variables

In Pascal, you declare boolean variables the same as in standard Pascal. Both of the following are valid boolean variables:

This example declares the variables `cloudy` and `sunny` as boolean.

```
var
   cloudy: boolean;
   sunny: boolean;
```

Boolean Initialization

To initialize a boolean variable when you declare it in the `var` declaration of your program, use an assignment statement as follows:

This example initializes `cloudy` to true and `sunny` to false.

```
var
    cloudy: boolean := true;
    sunny: boolean := false;
```

You can also initialize boolean variables in the `var` declaration of a procedure or function; however, when you do so, you must also declare the variable as `static`:

This code initializes the variable `rainy` to false, which has been declared as `static boolean`.

```
function weather (x: integer): boolean;

var
    rainy: static boolean := false;
```

Boolean Constants

You declare boolean constants in Pascal the same as in standard Pascal. Three valid boolean constants follow:

This example declares the constants a as true and b as false. It also declares n as the value `odd(y)`.

```
const
    a = true;
    b = false;
    y = 15;
    n = odd(y);
```

Data Representation

Pascal allocates one byte for each boolean variable. Figure 2-5 shows how Pascal internally represents a true boolean variable, and Figure 2-6 shows how Pascal represents a false boolean variable.

Figure 2-5 True Boolean Variable

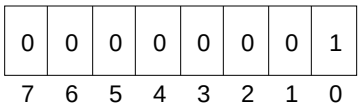
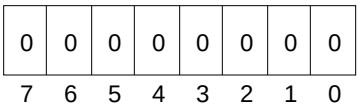


Figure 2-6 False Boolean Variable



Character

Pascal supports the standard predeclared data type `char`. As extensions to the standard, Pascal supports character initialization in the variable declaration and four nonstandard character constants.

Character Variables

You declare character variables in Pascal the same as you do in standard Pascal. Each of the following is a valid character variable:

```
var
    current_character: char;
    largest: char;
    smallest: char;
```

Character Initialization

To initialize a character variable when you declare it in the `var` declaration of your program, create an assignment statement as follows:

This example initializes the variable `pass` to A and `fail` to F.

```
var
    pass: char := 'A';
    fail: char := 'F';
```

You can also initialize character variables in the `var` declaration of a procedure or function; however, when you do so, you must also declare the variable as `static`:

This example initializes the variable `grade1` to A, `grade2` to B, and `grade3` to C. All three variables are declared as `static char`.

```
procedure grades;
var
    grade1: static char := 'A';
    grade2: static char := 'B';
    grade3: static char := 'C';
```

Character Constants

Pascal extends the standard definition of character constants by predeclaring the four character constants in Table 2-6.

Table 2-6 Nonstandard Predeclared Character Constants

Constant	Description
`minchar`	Equal to `char(0)`
`maxchar`	Equal to `char(255)`
`bell`	Equal to `char(7)` (which makes your terminal beep)
`tab`	Equal to `char(9)` (which makes a tab character)

Data Representation

Pascal allocates one byte for each character variable.

2

Enumerated

Pascal supports enumerated data types with extensions that allow you to input enumerated types with the `read` and `readln` procedures and output them with the `write` and `writeln` procedures. See the listings on `read` and `write` in Chapter 7, “Input and Output for details.

Enumerated Variables

You declare enumerated data types in Pascal the same as in standard Pascal.

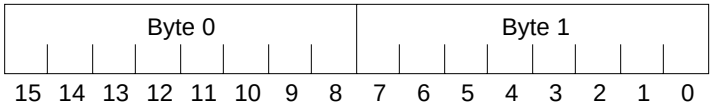
```
type
    continents =(North_America, South_America,
                 Asia, Europe, Africa, Australia,
                 Antartica);
    gem_cuts = (marquis, emerald, round, pear_shaped);

var
    x: gem_cuts;
    index: continents;
```

Data Representation

When you compile your program without the `–xl` option, Pascal represents enumerated types as either 8 or 16 bits, depending on the number of elements defined for that type. With `-xl`, Pascal represents variables of enumerated type as 16 bits. Pascal stores enumerated types as integers corresponding to their ordinal value.

Figure 2-7 shows the representation of a 16-bit enumerated variable.

Figure 2-7 16-Bit Enumerated Variable

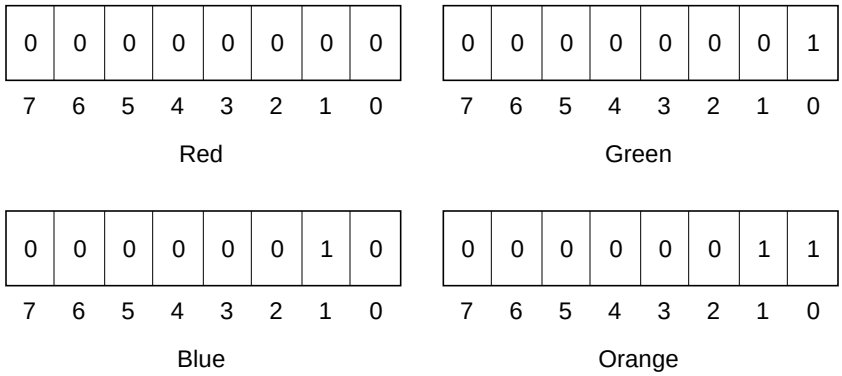


As an example, suppose you defined a group of colors as follows:

```
colors = (red, green, blue, orange);
```

Pascal represents each value as shown in Figure 2-8.

Figure 2-8 Sample Enumerated Representation



Subrange

Pascal supports a subrange of integer, boolean, character, and enumerated data types.

The Pascal subrange type is extended to allow constant expressions in both the lower and upper bound of the subrange. The lower bound expression is restricted by requiring that the expression not begin with a left parenthesis.

Subrange Variables

See the section of this chapter on unsigned integers for an example of a subrange declaration.

Data Representation

The Pascal subrange takes up the number of bits required for its minimum and maximum values. Table 2-7 shows the space allocation of six subranges.

Table 2-7 Subrange Data Representation

Min/Max range	Without –xl	With –xl
0..127	8 bits	16 bits
-128..127	8 bits	16 bits
0..255	16 bits	16 bits
-32768..32767	16 bits	16 bits
0...65536	32 bits	32 bits
-2,147,483,648..2,147,483,647	32 bits	32 bits

Figure 2-9 shows how Pascal represents a 16-bit subrange. Similarly, Figure 2-10 shows how Pascal represents a 32-bit subrange.

Figure 2-9 16-Bit Subrange

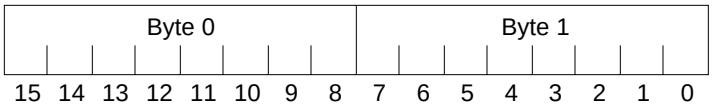
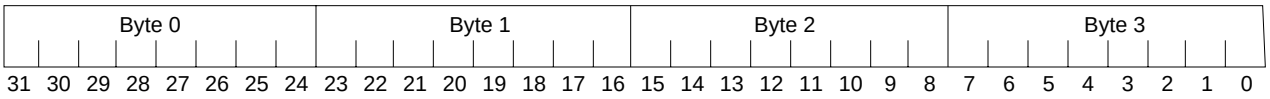


Figure 2-10 32-Bit Subrange



Record

Pascal supports the standard `record` and `packed record` data types. As an extension, Pascal permits you to initialize a record variable when you declare it in the variable declaration.

Record Variables

You declare records in Pascal the same as in standard Pascal, as shown in the following example:

```
type
    MonthType = (Jan, Feb, Mar, Apr, May, Jun, Jul,
    Aug, Sep, Oct, Nov, Dec);
    DateType = record
          Month : MonthType;
          Day : 1..31;
          Year : 1900..2000;
    end;

    Appointment = record
          Date : DateType;
          Hour : 0..2400;
    end;
```

Record Initialization

To initialize a field in a record when you declare it in the `var` declaration of your program, use either of the following two formats:

- Specify the record field name followed by an assignment operator and initial value.

```
 [a := FALSE ,
 b := TRUE ]
```

- List the initial value without the field name. In this case, Pascal assigns the initial value to the next field name in the record definition.

```
[  FALSE  ,
TRUE   ]
```

You can also initialize record variables in the `var` declaration of a procedure or function; however, when you do so, you must also declare the variable as `static`.

Example
The Pascal program, `init_rec.p`. This example shows a record initialization by name, by position, and by name and position.

```
program init_rec(output);

{ This program initializes a record. }

type
    enumerated_type = (red, green, blue, orange, white);
    record_type =
      record
          c: char;
          st: set of char;
          z: array [1..10] of char;
          case colors: enumerated_type of
              red: ( b: boolean;
                     s: single );
              green: ( i16: integer16;
                       d: double )
      end;
var
    { Initialization by name. }
    rec1: record_type :=
      [st := ['a', 'b', 'c'],
      c := 'A',
      z := 'ARRAY1',
      colors := green,
      i16 := 32767];
```

Initializing Record Variables (Screen 1 of 2)

```
    { Initialization by position. }
    rec2: record_type :=
      ['X',
      ['x', 'y', 'z'],
      'ARRAY2',
      red,
      true];
    { Initialization by name and position. }
    rec3: record_type :=
      [colors := red,
      true,
      1.16,
      st := ['m', 'n', 'o'],
      'ARRAY3'];

begin
    writeln('char       ', rec1.c);
    writeln('char array ', rec1.z);
    writeln('integer    ', rec1.i16);
    writeln;
    writeln('char       ', rec2.c);
    writeln('char array ', rec2.z);
    writeln('boolean    ', rec2.b);
    writeln;
    writeln('char array ', rec3.z);
    writeln('boolean    ', rec3.b);
    writeln('single     ', rec3.s)
end. { record_example }
```

Initializing Record Variables (Screen 2 of 2)

The commands to compile and execute `init_rec.p`:

```
hostname% pc init_rec.p
hostname% a.out
char       A
char array ARRAY1
integer         32767

char       X
char array ARRAY2
boolean    true

char array ARRAY3
boolean    true
single      1.160000e+00
hostname%
```

Data Representation of Unpacked Records

This section describes the data representations of unpacked fixed and variant records.

Fixed Records

Pascal allocates fields in a fixed record so that they assume the natural alignment of the field type. The alignment of a record is equal to the alignment of the largest element. The size of the record is a multiple of the alignment.

Variant Records

The space Pascal allocates for a variant record is the same with or without the `-xl` option.

Data Representation of Packed Records

Table 2-8, Table 2-9 and Table 2-10 show how Pascal aligns fields in a packed record. (Note that in packed records bit aligned fields do not cross word boundaries.)

Packed Record Storage without the `-xl` *Option*

Table 2-8 Packed Record Storage Without `-xl`

Data Type	Size	Alignment
`integer`	4 bytes	4 bytes
`integer16`	2 bytes	Bit-aligned
`integer32`	4 bytes	4 bytes
`real`	8 bytes	8 bytes
`single`	4 bytes	4 bytes
`shortreal`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`longreal`	8 bytes	8 bytes
`boolean`	1 bit	Bit-aligned
`char`	1 byte	1 byte
enumerated	Number of bits required to represent the highest ordinal value.	Bit-aligned
subrange of `char`	1 byte	1 byte
all other subrange	Number of bits required to represent the highest ordinal value.	Bit-aligned
set of cardinality <= 32	1 bit per element	Bit-aligned
set of cardinality > 32	Same as if unpacked.	4 bytes
array	Requires the same space required by the base type of the array.	Same as element type

Packed Record Storage with the `-xl` *Option*

Table 2-9 Packed Record Storage With `–xl`

Data Type	Size	Alignment
`real`	4 bytes	4 bytes
`integer`	2 bytes	Bit-aligned

Packed Record Storage with the `-calign` *Option*

Table 2-10 Packed Record Storage With `-calign`

Data Type	Size	Alignment
`integer16`	2 bytes	2 bytes

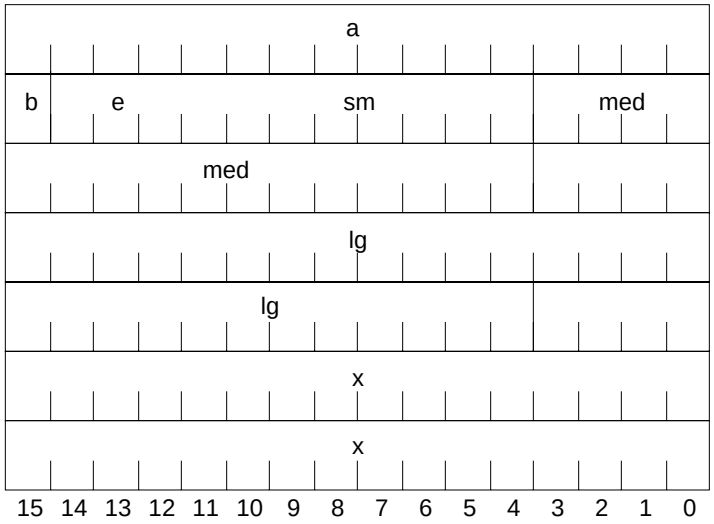
The following example declares a packed record. Table 2-11 shows the alignment and sizes of the fields of the record. Figure 2-11 shows the representation of this record.

```
type
    small = 0..128;
    medium = 0..255;
    large = 0..65535;
    colors = (green, blue, orange, white, black,
                      magenta, gray);
    sets = (autumn, summer, winter, fall);
    vrec1 = packed record
       a: integer16;
       b: boolean;
       e: colors;
       sm: small;
       med: medium;
       lg: large;
       se: sets;
       x: integer32;
    end;
```

Table 2-11 Sample Sizes and Alignment of Packed Record

Field	Size	Alignment
a	16 bits	16 bit-aligned
b	1 bit	Bit-aligned
e	3 bits	Bit-aligned
sm	8 bits	Bit-aligned
med	16 bits	Bit-aligned
lg (without -xl)	32 bits	32 bit-aligned
lg (with -xl)	16 bits	16 bit-aligned
se	4 bits	Bit-aligned
x	32 bits	32 bit-aligned

Figure 2-11 Sample Packed Record (Without -xl)



Array

Pascal supports the standard array data type. As extensions to the standard, Pascal supplies the predeclared character array types alfa, string, and varying and permits you to initialize an array variable when you declare it in the variable declaration.

Array Variables

In addition to the standard array data types, this compiler supports the three data types in Table 2-12, which include a variable-length string.

Table 2-12 Array Data Types

Type	Description
`alfa`	An array of char 10 characters long.
`string`	An array of char 80 characters long.
`varying`	A string of variable length. You declare a varying string as follows: `varying`[*upper_bound*] `of char`; *upper_bound* is an integer between 0 and 65535.

You can assign a variable-length string a string of any length, up to the maximum length you specify in the declaration. Pascal ignores any characters you specify over the maximum. It does not pad the unassigned elements with spaces if you specify a string under the maximum. When you output a variable-length string with `write` or `writeln`, the procedure writes only the characters included in the string's current length.

You also can assign a variable-length string to a fixed-length string. If the variable-length string is shorter than the fixed-length string, the fixed-length string is padded with blanks. If the variable-length string is longer than the fixed-length string, the string is truncated.

This program demonstrates the differences between the fixed-length and `varying` data types:

Example
The Pascal program, `varying.p`:

```
program varying_example(output);

{ This program demonstrates the differences
  between fixed- and variable-length strings. }

var
    name1: array [1..25] of char;  { String of size 25. }
    name2: array [76..100] of char;{ String of size 25. }
    name3: alfa;                   { String of size 10. }
    name4: string;                 { String of size 80. }
    name5: varying [25] of char;   { Varying string. }
    name6: varying [25] of char;   { Varying string. }

begin
    name1 := 'van Gogh';
    name2 := 'Monet';
    name3 := 'Rembrandt';
    name4 := 'Breughel';
    name5 := 'Matisse';
    name6 := 'Cezanne';
    writeln(name1, ' and ', name2, '.');
    writeln(name3, ' and ', name4, '.');
    writeln(name5, ' and ', name6, '.')
end. { varying_example }
```

The commands to compile and execute `varying.p`:

```
hostname% pc varying.p
hostname% a.out
van Gogh                  and Monet                     .
Rembrandt  and
Breughel                                                .
Matisse and Cezanne.
hostname%
```

Array Initialization

To initialize an array variable when you declare it in the `var` declaration of your main program, use an assignment statement after the declaration. Pascal offers you four different formats:

- Supply the lower and upper bounds in the initialization.

This code initializes the first five elements of `int` to `maxint`, 1, -32767, 5, and 20. The first six elements of `c1` are assigned the characters '1' through '6'. Because `c1` is a fixed-length string, the last four characters are padded with blanks.

```
var
    int : array[1..10] of integer := [maxint, 1, -327, 5, 20];
    c1 :  array[1..10] of char := '123456';
```

- Put an asterisk in place of the upper bound, and let the compiler determine the upper bound once it counts the initial values. You may only use this format when you also supply the initial values.

In this example, the compiler assigns the upper bound of 5 to `int` and of 6 to `c1`.

```
var
 i : integer;
 int : array[1..*] of integer := [maxint, 1, -32767, 5, 20];
 c1 : array[1..*] of char := '123456';
```

- Use the repeat count feature *n* `of` *constant* to initialize *n* array elements to the value *constant*. *n* must be an integer or an expression that evaluates to an integer constant.

This code initializes all the first 50 values of `int2` to 1 and the second 50 values to 2.

```
var
    int2 : array[1..100] of integer := [50 of 1, 50 of 2];
```

- Use the repeat count feature `*` `of` *constant* to initialize all remaining array elements to the value of *constant*.

This example initializes all 100 elements of `int4` to 327. The example also initializes the multidimensional array `int5` to an array of 10 rows and columns. The compiler initializes all 10 elements in the first row to 327, the first three elements of the second row to 8, and all 10 elements of the third row to 88.

```
var
    int4 : array[1..100] of integer := [* of 327];
    int5 : array[1..10,1..10] of integer := [
                                        [* of 327],
                                        [3 of 8],
                                        [10 of 88],
                                        ];
```

When you initialize an array in the `var` declaration, the compiler sets those elements for which it doesn't find data to zero.

You can also initialize array variables in the `var` declaration of a procedure or function; however, you must also declare the variable as `static`.

Packed Arrays

Although you can define an array as `packed`, it will have no effect on how the Pascal compiler allocates the array data space.

Data Representation

The elements of an array require the same space as that required by the base type of the array. However, there are two exceptions to this. With the `-calign` option, the size of all arrays is the size of each element times the number of elements. When you use the `-calign` and `-xl` options together, arrays are the same as with `-calign` alone, except the size of an array of booleans is always a multiple of two.

Set

Pascal supports sets of elements of integer, boolean, character, and enumerated data types. As extensions to the standard, Pascal predefines a set of `intset` and allows you to initialize a set variable when you declare it in the `var` declaration of your program.

Set Variables

In Pascal, you declare set variables the same as you do in standard Pascal. The following is a valid set variable:

```
type
    character_set = set of char;

var
    letters: character_set;
```

intset

Pascal predefines the set `intset` as the set of [0..127].

Set Initialization

To initialize a set variable when you declare it in the `var` declaration of your program, create an assignment statement as follows:

This code initializes `citrus` to the set of `orange`, `lemon`, and `lime`.

```
type
  fruit = (orange, lemon, lime, apple, banana);
var
  citrus: set of fruit := [orange, lemon, lime];
```

You can also initialize set variables in the `var` declaration of a procedure or function; however, when you do so, you must also declare the variable as `static`:

This example initializes `primary` to the set of `red`, `yellow`, and `blue`. It also initializes `grays` to the set of `white` and `black`.

```
procedure assign_colors;
type
  colors = (white, beige, black, red, blue,
               yellow, green);
var
  primary: static set of colors := [red, yellow,
               blue];
  grays: static set of colors := [white, black];
```

Packed Sets

Although you can define a set as `packed`, it will have no effect on how the compiler allocates the set data space.

Data Representation

Pascal implements sets as bit vectors, with one bit representing each element of a set. The maximum ordinal value of a set element is 32768.

The size of a set is determined by the size of the ordinal value of maximal element of the set plus one. Sets are allocated in multiples of 16 bits; therefore, the smallest set has size 16 bits. The ordinal value of the minimal element must be equal to or greater than 0. Sets have an alignment of four bytes when they are longer than 16 bits; otherwise their alignment is two bytes. For example, 'set of 1..20' has a four byte alignment and 'set of 1..15' has a two byte alignment.

With the `-calign` option, sets have an alignment of two bytes. The size is the same as the default.

Table 2-13 shows the data representation of four sets.

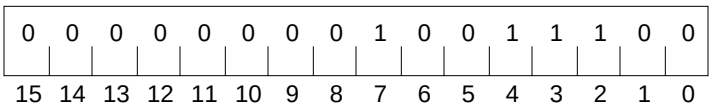
Table 2-13 Data Representation of Sets

Set	Description
set of 0..15	This set requires 16 bits because 15 is the maximal element, and 15 + 1 = 16.
set of 0..16	This set requires 32 bits because 16 is the maximal element. 16 + 1 = 17, and the next multiple of 16 above 17 is 32.
set of 14..15	This set requires 16 bits because 15 is the element, and 15 + 1 = 16.
set of char	This set requires 256 bits because the range of `char` is `chr(0)..chr(255)` . The ordinal value of the maximal element is 255, and 255+1 = 256, which is divisible by 16.

You can visualize the bit vector representation of a set as an array of bits starting from the highest element to the lowest element. For example, the representation of the following set is shown in Figure 2-12.

```
var
   smallset: set of 2..15 := [7,4,3,2];
```

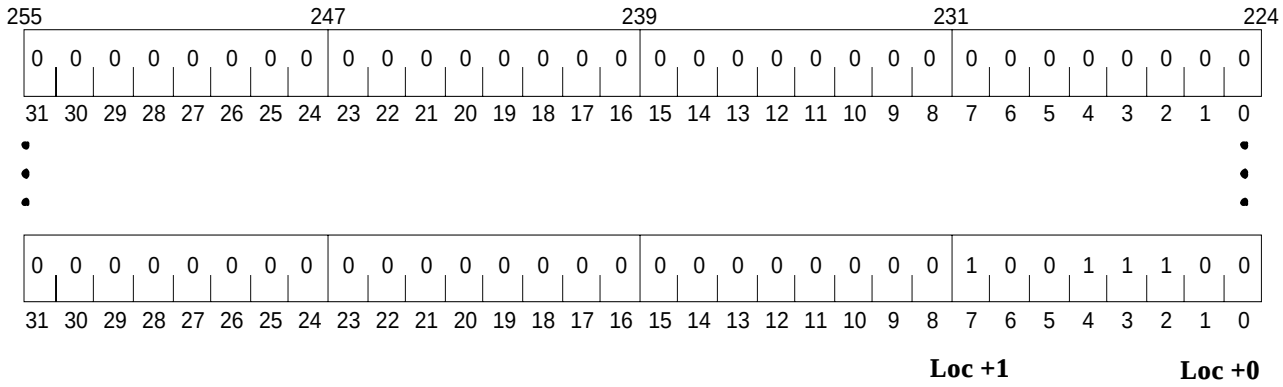
Figure 2-12 Small Set



The representation of this larger set is shown in Figure 2-13.

```
var
    largeset: set of 2..255 := [7,4,3,2];
```

Figure 2-13 Large Set



File

Pascal treats files declared as `file of char` the same as files declared as `text`, except when you use the `–s` `–s0`, `–s1` `–V0` or `–V1` options. In this case, the compiler treats the two data types differently, as required by standard Pascal.

Pointer

Pascal supports the standard Pascal pointer and the nonstandard universal pointer and procedure and function pointer.

Standard Pointer

The standard pointer is the same in Pascal and standard Pascal.

Universal Pointer

The universal pointer data type, `univ_ptr`, is compatible with any pointer type. Use `univ_ptr` to compare a pointer of one type to another or to assign a pointer of one type to another.

You cannot dereference a `univ_ptr` variable: you cannot find the contents of the address to which `univ_ptr` points.

Example
The Pascal program, `univ_ptr.p`, which prints the value of the floating-point variable `r` in hexadecimal format.

```
program univ_ptr_example;

{ This program demonstrates how to use
  universal pointers. }

var
    i: integer32;
    r: single;
    ip: ^ integer32;
    rp: ^ single := addr(r);
    up: univ_ptr;

begin
    r := 10.0;
    { The next two statements are equivalent to rp := ip.
      However, rp := ip is not legal since they are
      different types. }
    up := rp;
    ip := up;
    writeln(ip^ hex);
    { This will do the same thing but uses transfer functions. }
    writeln(integer32(r) hex)
end. { univ_ptr_example }
```

The commands to compile and execute `univ_ptr.p`:

```
hostname% pc univ_ptr.p
hostname% a.out
41200000
41200000
hostname%
```

Procedure and Function Pointers

The syntax for procedure and function pointers follows:

```
<procedure pointer>  ::= '^' <procedure heading>
        |  <type identifier>
```

```
<function pointer>  ::= '^' <function heading>
        |  <type identifier>
```

Example
The Pascal program, `pointer.p`. This program demonstrates how to print out enumerated values using procedure pointers.

```
program pointer_example;

type
    colors = (red, white, blue);
    procptr = ^ procedure; { Procedure pointer type. }

procedure printred;

begin
    writeln('RED')
end; { printred }

procedure printwhite;

begin
    writeln('WHITE')
end; { printwhite }

procedure printblue;

begin
    writeln('BLUE')
end; { printblue }

var
    { Array of procedure pointers. }
    colorprinter: array [colors] of procptr :=
              [addr(printred),
              addr(printwhite),
              addr(printblue)];
    c: colors;
    desc_proc: procptr;

begin
    write('Enter red, white, or blue:  ');
    readln(c);
    desc_proc := colorprinter[c];
    desc_proc^
end. { pointer_example }
```

The commands to compile and execute `pointer.p`:

```
hostname% pc pointer.p
hostname% a.out
Enter red, white, or blue:  red
RED
hostname%
```

Pointer Initialization

To initialize a pointer variable when you declare it in the `var` declaration of your program, use an assignment statement as follows:

This example initializes the variable `rp` to `addr(r)`. Legal values for compile-time initializations are `NIL`, `addr(0)` of variables, procedures, strings, and set constants, and previously declared constants of the same pointer type.

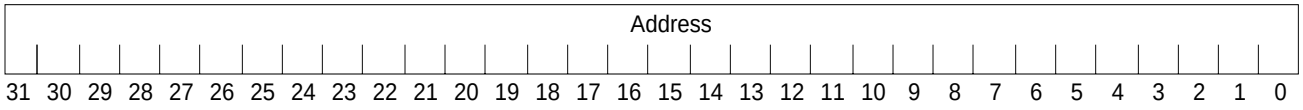
```
var
   rp : ^single := addr(r);
   pp : ^procedure := NIL;
   sp : ^string := addr('Title');
```

You can also initialize pointer variables in the `var` declaration of a procedure or function; however, when you do so, you must also declare the variable as `static`.

Data Representation

Pascal represents a pointer as shown in Figure 2-14.

Figure 2-14 Pointer



2

Statements 3

This chapter describes Pascal statements.

Standard Statements

Pascal supports all standard statements. Pascal also supports extensions to `case`, `goto`, and `with`, which are described in the following section.

Statements Specific to Pascal

Table 3-1 summarizes the nonstandard Pascal statements and standard statements with nonstandard features. The following sections contain detailed descriptions and examples of each statement.

3

Table 3-1 Non-Standard Pascal Statements

Statement	Description
`assert`	Causes a boolean expression to be evaluated each time the statement is executed.
`case`	Accepts ranges of constants and an `otherwise` statement.
`exit`	Transfers program control to the first statement after the end of a `for`, `while`, or `repeat` loop.
`goto`	Accepts an identifier as the target of goto.
`next`	Causes the program to skip to the next iteration of the enclosing `for`, `while`, or `repeat` loop.
`otherwise`	An extension to the `case` statement. If the expression in a `case` statement does not match any of the `case` values, the compiler executes the statements under the `otherwise` statement.
`return`	Prematurely ends a procedure or a function.
`with`	An alternative format to the standard `with` statement.

`assert` *Statement*

The `assert` statement causes a boolean expression to be evaluated each time the statement is executed.

<*assert statement*> : : = ’assert’ ’(’ <*boolean expression*> ’)’

If your program contains an `assert` statement, you must compile it with the `–C` option, which enables runtime tests. Otherwise, the compiler treats `assert` as a comment.

A runtime error results if the expression in the `assert` statement evaluates to false.

`assert` is a shorthand for using the `if` statement. For example, the following code tests whether `num` is greater than 0 and less than or equal to `MAX_STUDENTS` using an `assert`:

```
assert((num > 0) and (num <= MAX_STUDENTS));
for i := 1 to num do begin
    write('Enter grade for student ', i: 3, ': ');
    readln(grades[i])
end
```

The following `if` statement is equivalent to the `assert` statement in the preceding program:

```
if (num > 0) and (num <= MAX_STUDENTS) then begin
    for i := 1 to num do begin
          write('Enter grade for student ', i: 3, ': ');
          readln(grades[i])
    end
end else begin
    writeln('Error message.');
    halt
end
```

Example
The Pascal program, `assert.p`, which tests whether `num` is greater than 0 and less than or equal to `MAX_STUDENTS` before reading in the grades.

```
program assert_example;

const
    MAX_STUDENTS = 4;

var
    num: integer;
    i: integer;
    grades: array [1..MAX_STUDENTS] of char;

begin
    num := 6;
    assert((num > 0) and (num <= MAX_STUDENTS));
    for i := 1 to num do begin
        write('Enter grade for student ', i: 3, ':  ');
        readln(grades[i])
    end
end. { assert_example }
```

The commands to compile and execute `assert.p` without the **`-C`** option. The compiler treats `assert` as a comment.

```
hostname% pc assert.p
hostname% a.out
Enter grade for student   1:  A
Enter grade for student   2:  B
Enter grade for student   3:  C
Enter grade for student   4:  D
Enter grade for student   5:  F
Enter grade for student   6:  A
hostname%
```

The result when you compile `assert.p` with the `-C` and `-g` option. The expression evaluates to `false`, so the compiler generates an error and halts.

```
hostname% pc -C assert.p
hostname% a.out

Assertion #1 failed
Trace/BPT trap (core dumped)

hostname% pc -C -g assert.p
hostname% a.out

Assertion #1 failed
Trace/BPT trap (core dumped)
hostname%
```

`case` *Statement*

Pascal supports the standard `case` statement with extensions for an `otherwise` clause and ranges of constants.

```
<case statement>  ::= 'case' <expression> 'of'
    { <case values> ':' <statement> ';' }
    [ 'otherwise' <statement list> ]
    'end'
```

```
<case values> ::= <case range> { ',' <case range> }
```

```
<case range> ::= <constant expression>
     | <constant expression> '..' <constant expression>
```

If *expression* does not match any of the *case values*, the compiler executes the `otherwise` *statement list*. The reserved word `otherwise` is not a case label, so it is not followed by a colon (:). Also, the `begin/end` pair is optional in an `otherwise` statement.

You may use a range of constants instead of a single `case` value. A `case` range must be in ascending order.

The `case` statement operates differently when you compile your program with and without the `–xl` option. Without `–xl`, if the value of the expression is not equal to one of the `case` labels and you omit the `otherwise` statement, the program generates an error and halts. If this situation occurs and you compile your program with `–xl`, the program falls through and does not generate an error; program execution continues with the statement immediately following the `case` statement.

Example
The Pascal program, `otherwise.p`. This program reads a character from the terminal. If the value of the character is not in the range 0 - 9, the compiler executes the statement in the otherwise statement. Note that the program specifies all digits between 0 and 9 as the range '0'..'9' .

```
program otherwise_example(input, output);

{ This program demonstrates the otherwise
  clause and ranges in the case statement. }

var
    ch: char;

begin
    write('Please enter one character:  ');

{More than one character will produce erroneous results.}
    readln(ch);
    case ch of
      '0'..'9':
        writeln('The character you input is a digit.');
 otherwise
             writeln('The character you input is not a digit.')
    end
end. { otherwise_example }
```

The commands to compile and execute `otherwise.p` without **`–xl`**. This example shows your output when you input the characters 3 and B.

```
hostname% pc otherwise.p
hostname% a.out
Please enter one character:  3
The character you input is a digit.
hostname% a.out
Please enter one character:  B
The character you input is not a digit.
hostname%
```

`exit` *Statement*

The `exit` statement, which you can use in a `for`, `while`, or `repeat` loop, transfers program control to the first statement after the end of the current loop.

```
<exit statement> ::= 'exit'
```

If used in a nested loop, `exit` only breaks out of the innermost loop.

You will receive a compile-time error if you use this statement anywhere but in a `for`, `while`, or `repeat` loop.

Example
The Pascal program, `exit.p`:

```
program exit_example(input, output);

{ This program demonstrates the use of the
  exit statement in for, while, and repeat loops. }

const
    MAX = 10;

type
    integer_type = array [1..MAX] of integer16;

var
    i: integer16;
    i_array: integer_type := [1, 99, 13, 45, 69, 18, 32, -6];
    number: integer16;
    flag: boolean := false;

begin
    write('Enter a number:  ');
    readln(number);
    for i := 1 to MAX do begin
      if number = i_array[i] then begin
          flag := true;
          exit
      end
    end;
    if flag then
      writeln('Number WAS found:  ', number)
    else
      writeln('Number WAS NOT found:  ', number)
end. { exit_example }
```

The commands to compile and execute `exit.p`. This example shows the program output when you input the number 13.

```
hostname% pc exit.p
hostname% a.out
Enter a number:  13
Number WAS found:          13
hostname%
```

goto *Statement*

Pascal supports the standard format of the goto statement with two extensions.

```
<goto statement> ::= 'goto' <label>
```

```
<label> ::= <number> | <identifier>
```

In Pascal, you can use an identifier as the target of a goto. Standard Pascal allows only integers as targets of gotos.

If you use a goto to jump out of the current block, Pascal closes all open files in the intervening blocks between the goto statement and the target of the goto.

Example
The Pascal program, `goto.p`, which uses an identifier as a target of a `goto` statement.

```
program goto_example;

{ This program uses an identifier as a target
  of a goto statement. }

label
    skip_subtotal;

const
    MAX_STUDENTS = 100;

var
    i: integer;
    grades: array [1..MAX_STUDENTS] of char;
    num: 1..MAX_STUDENTS;
    sum: real;
    points: real;

begin
    { Read in number of students and their grades. }
    write('Enter number of students:  ');
    readln(num);
    assert((num > 0) and (num < MAX_STUDENTS));
    for i := 1 to num do begin
      write('Enter grade for student ', i: 3, ': ');
      readln(grades[i])
    end;
    writeln;
 { Now calculate the average GPA for all students. }
    sum := 0;
    for i := 1 to num do begin
      if grades[i] = 'I' then begin
          goto skip_subtotal
      end else begin
          case grades[i] of
              'A': points := 4.0;
              'B': points := 3.0;
              'C': points := 2.0;
              'D': points := 1.0;
              'F': points := 0.0;
```

Identifiers as Targets (Screen 1 of 2)

```
    otherwise
              writeln('Unknown grade:  ', grades[i]);
              points := 0.0
          end
      end;
      sum := sum + points;
      skip_subtotal:
    end;
    writeln('GPA for all students is ', sum / num: 6: 2, '.')
end. { goto_example }
```

Identifiers as Targets (Screen 2 of 2)

You must compile `goto.p` with the `-C` option to execute the `assert` statement; otherwise, the compiler treats `assert` as a comment. This example returns the collective GPA of four

```
hostname% pc -C goto.p
hostname% a.out
Enter number of students:    4
Enter grade for student   1: B
Enter grade for student   2: B
Enter grade for student   3: C
Enter grade for student   4: A

GPA for all students is   3.00.
hostname%
```

`next` *Statement*

The `next` statement, which you can only use in a `for`, `while`, or `repeat` loop, causes the program to skip to the next iteration of the current loop, thus skipping the rest of the statements in the loop.

```
<next statement> ::= 'next'
```

The `next` statement has the same effect as a `goto` to the end of the loop. If you use `next` in a `for` loop, Pascal increments the index variable as normal.

When you use `next` in a nested loop, it goes to the end of the innermost loop containing the `next` statement.

You will receive a compile-time error if you use this statement anywhere but in a `for`, `while`, or `repeat` loop.

Example
The Pascal program, `next.p`. This program also uses the `otherwise` statement discussed earlier in this chapter.

```
program next_example;

{ This program demonstrates the use of the next
  statement in for, while, and repeat loops. }

const
    MAX_STUDENTS = 100;

var
    i: integer;
    grades: array [1..MAX_STUDENTS] of char;
    num: 1..MAX_STUDENTS;
    sum: real;
    points: real;

begin
    { Read in number of students and their grades. }
    write('Enter number of students:  ');
    readln(num);
    assert((num > 0) and (num <= MAX_STUDENTS));
    for i := 1 to num do begin
      write('Enter grade for student ', i: 3, ':  ');
      readln(grades[i])
    end;
    writeln;
```

The `next` Statement (Screen 1 of 2)

```
    { Now calculate the average GPA for all students. }
    sum := 0;
    for i := 1 to num do begin
      if grades[i] = 'I' then begin
          next
      end else begin
          case grades[i] of
              'A': points := 4.0;
              'B': points := 3.0;
              'C': points := 2.0;
              'D': points := 1.0;
              'F': points := 0.0;
              otherwise
                  writeln('Unknown grade:  ', grades[i]);
                points := 0.0
                end
      end;
      sum := sum + points
    end;
     writeln('GPA for all students is:  ', sum / num: 6: 2)
end. { next_example }
```

The `next` Statement (Screen 2 of 2)

You must compile `next.p` with the `-C` option to execute the `assert` statement; otherwise, the compiler treats `assert` as a comment. This example outputs the collective GPA of three students.

```
hostname% pc -C next.p
hostname% a.out
Enter number of students:  3
Enter grade for student   1:  A
Enter grade for student   2:  A
Enter grade for student   3:  C

GPA for all students is:    3.33
hostname%
```

`otherwise` *Statement*

The `otherwise` statement is a Pascal extension to the standard Pascal `case` statement. If specified, `otherwise` must be at the end of the `case` statement. See the listing on the `case` statement earlier in this chapter for additional information.

`return` *Statement*

The `return` statement prematurely ends a procedure or a function.

```
<return statement> ::= 'return'
```

Program control transfers to the calling routine. This has the same effect as a `goto` to the end of the routine. If used in the main program, `return` causes the program to terminate.

Example
The Pascal program, `return.p`. The compiler prematurely returns from the procedure `test` if you input 1 or any integer from 4 through 99. The program also uses identifiers as the target of a `goto`.

```
program return_example;

{ This program demonstrates the use of the
  return statement in a procedure. }

var
    i: integer;

procedure test;
label
    error_negative_value, error_bad_values, error_value_too_big;
begin
    if i < 0 then
      goto error_negative_value
    else if (i = 2) or (i = 3) then
      goto error_bad_values
    else if i > 100 then
      goto error_value_too_big;
    return;
error_negative_value:
    writeln('Value of i must be greater than 0.');
    return;
error_bad_values:
    writeln('Illegal value of i:  2 or 3.');
    return;
error_value_too_big:
    writeln('Value of i too large.');
    return
end; { test }

begin { main procedure }
    write('Enter value for i:  ');
    readln(i);
    test
end. { return_example }
```

The commands to compile and execute `return.p`:

```
hostname% pc return.p
hostname% a.out
Enter value for i:  -1
Value of i must be greater than 0.
hostname% a.out
Enter value for i:  2
Illegal value of i:  2 or 3.
hostname% a.out
Enter value for i:  101
Value of i too large.
hostname% a.out
Enter value for i:  5
hostname%
```

`with` *Statement*

Pascal supports the standard with statement plus an alternative format.

<*with statement*> `::= 'with'` <*record variable list*> `'do'` <*statement*>

<*record variable list*> `::=` <*record variable*> { `' ,'` <*record variable*> }

<*record variable*> *::= variable [':'* <*with_identifier*>*]*

<*with_identifier*> `::=` <*identifier*>

This form of the `with` statement is used as an alias or shorthand notation for a qualified variable. The scope of the *with_identifier* is the lexical scope of the `with` statement.

Example
The Pascal program, `with.p`, which uses the alternate form of the `with` statement.

```
program with_example(output);

{ Sample program using the extension to the
  with statement. }

const
    MAX = 12;

type
    name_type = varying [MAX] of char;
    Patient =
      record
          LastName: name_type;
          FirstName: name_type;
          Sex: (Male, Female)
      end;

var
    new_patient: Patient;
    old_patient: Patient;

begin
    with new_patient: new, old_patient: old do begin
      new.LastName := 'Smith';
      new.FirstName := 'Abby';
      new.Sex := Female;

      old.LastName := 'Brown';
      old.FirstName := 'Henry';
      old.Sex := Male
    end;
    write('The new patient is ');
    write(new_patient.FirstName: 10);
    writeln(new_patient.LastName: 10, '.');
    write('The old patient is ');
    write(old_patient.FirstName: 10);
    writeln(old_patient.LastName: 10, '.')
end. { with_example }
```

The commands to compile and execute `with.p`:

```
hostname% pc with.p
hostname% a.out
The new patient is       Abby     Smith.
The old patient is      Henry     Brown.
hostname%
```

Assignments and Operators 4

This chapter describes assignments and operators in Pascal.

Data Type Assignments/Compatibility

Table 4-1 lists the assignment compatibility rules for real, integer, boolean, character, enumerated, subrange, record, set, and pointer data types.

Table 4-1 Data Type Assignments

Type of Variable/Parameter	Type of Assignment-Compatible Expression
`real`, `single`, `shortreal`	`real`, `single`, `shortreal`, `double`, `longreal`, any integer type†
`double`, `longreal`	`real`, `single`, `shortreal`, `double`, `longreal`, any integer type††
`integer`, `integer16` `integer32`	`integer`, `integer16`, `integer32`
`boolean`	`boolean`
`char`	`char`
enumerated	Same enumerated type
subrange	Base type of the subrange
record	Record of the same type
array	Array with structurally compatible type††
set	Set with compatible base type
pointer	Pointer to a structurally compatible type, `univ_ptr`

† Pascal implicitly converts the integer to the real type, if necessary.

†† The exception is the `array of char`, described in the next section.

String Assignments

Pascal has special rules for assigning fixed- and variable-length strings, null strings, and string constants.

Fixed- and Variable-Length Strings

When you make an assignment to a fixed-length string, and the source string is shorter than the destination string, the compiler pads the destination string with blanks. If the source string is larger than the destination string, the compiler truncates the source string to fit the destination.

When you make an assignment to a variable-length string, and the source string is longer than the destination string, the compiler truncates the source to fit the destination.

The valid fixed- and variable-length string assignments are given in Table 4-2.

Table 4-2 Fixed- and Variable-Length String Assignments

Type of String	**Type of Assignment-Compatible Expression**
`array of char`	`varying` string, constant string, and `array of char` if the arrays have the same length.
`varying`	`varying` string, constant string, `array of char`, and `char`.

Null Strings

Pascal treats null strings as constant strings of length zero. Table 4-3 shows the null string assignments.

Table 4-3 Null String Assignments

Assignment	Description
`varying := '';`	The compiler assigns the null string to the variable-length string. The length of the variable-length string equals zero.
`array of char := '';`	The compiler assigns a string of blanks to the character array. The length of the resulting string is the number of elements in the source character array.
`char := '';`	It is illegal to assign a null string to a `char` variable. Use `chr(0)` instead.
String concatenation	In a string concatenation expression such as `S := 'hello' + '' + S;` `''` is treated as the additive identity (as nothing).

String Constants

When assigning a constant string to a packed `array of char`, standard Pascal requires that the strings be the same size.

Pascal allows the constant string and packed `array of char` to be unequal in size, truncating the constant string if it is longer or padding it with blanks if it is shorter.

Operators

Pascal supplies six classes of operators:

- Arithmetic operators
- Bit operators
- Boolean operators
- Set operators
- Relational operators

- String operators

Arithmetic Operators

The arithmetic operators are summarized in Table 4-4.

Table 4-4 Arithmetic Operators

Operator	Operation	Operands	Result
+	addition	integer or real	integer or real
-	subtraction	integer or real	integer or real
*	multiplication	integer or real	integer or real
/	division	integer or real	real
div	truncated division	integer	integer
mod	modulo	integer	integer

The mod *Operator*

Pascal extends the standard definition of the `mod` operator as follows. In the expression `i mod j`, when `i` is positive, Pascal and standard Pascal produce the same results. However, when `i` is negative, and you do not compile your program with a standard option (`-s`, `-s0`, `-s1`, `-V0`, or `-V1`), the following is true:

```
i mod j
```

equals:

```
     -1 * remainder of |i| divided by |j|
```

Example
The Pascal program, `mod.p`, which computes `i mod j`.

```
program modexample(output);

{ This program demonstrates the nonstandard
  mod function. }

var
    i: integer;
    j: integer;

begin
    for i := -3 to -1 do
      for j := 1 to 3 do
          if j <> 0 then
              writeln(i: 4, j: 4, i mod j: 4)
end. { mod_example }
```

The commands to compile and execute `mod.p` without any options.

```
hostname% pc mod.p
hostname% a.out
  -3   1   0
  -3   2  -1
  -3   3   0
  -2   1   0
  -2   2   0
  -2   3  -2
  -1   1   0
  -1   2  -1
  -1   3  -1
hostname%
```

The results negative `i` produces when you compile `mod.p` with the **`-s`** option.

```
hostname% pc -s mod.p
hostname% a.out
  -3   1   0
  -3   2   1
  -3   3   0
  -2   1   0
  -2   2   0
  -2   3   1
  -1   1   0
  -1   2   1
  -1   3   2
hostname%
```

Bit Operators

Table 4-5 shows the bit operators. The ~ operator produces the same results as the built-in Pascal function `lnot`. Similarly, & is equivalent to the function `land` and | and ! are equivalent to `lor`. See Chapter 7, "Input and Output for descriptions of these functions and the truth tables that both the functions and operators use.

Table 4-5 Bit Operators

Operator	Operation	Operands	Result
~	bitwise `not`	integer	integer
&	bitwise	integer	integer
|	bitwise `or`	integer	integer
!	bitwise `or` (same as |)	integer	integer

Boolean Operators

The `boolean` operators, which include the nonstandard `and then` and `or else` operators, are summarized in Table 4-6.

Table 4-6 Boolean Operators

Operator	Operation	Operands	Result
`and`	conjunction	`boolean`	`boolean`
`and then`	similar to boolean `and`	`boolean`	`boolean`
`not`	negation	`boolean`	`boolean`
`or`	disjunction	`boolean`	`boolean`
`or else`	similar to boolean `or`	`boolean`	`boolean`

The `and then` *Operator*

The `and then` operator differs from the standard `and` operator in that it guarantees the order in which the compiler evaluates the logical expression. Left to right and the right operands are evaluated only when necessary. For example, when you write the following syntax, the compiler may evaluate `odd(y)` before it evaluates `odd(x)`:

```
odd(x) and odd(y)
```

However, when you use the following syntax, the compiler always evaluates `odd(x)` first:

```
odd(x) and then odd(y)
```

If `odd(x)` is `false`, `odd(y)` is not evaluated.

Note – Comments cannot be inserted between the `and` and the `then` operators.

The Pascal program, `and_then.p`. This program uses `and then` to test if two numbers are odd.

```
program and_then(input, output);

{ This program demonstrates the use
  of the operator and then. }

var
    x, y: integer16;

begin
    write('Please enter two integers:  ');
    readln(x, y);
    if odd(x) and then odd(y) then
      writeln('Both numbers are odd.')
    else
      writeln('Both numbers are not odd.');
end. { and_then }
```

The commands to compile and execute `and_then.p`. This example shows the output when you input the numbers 45 and 6.

```
hostname% pc and_then.p
hostname% a.out
Please enter two integers:  45 6
Both numbers are not odd.
hostname%
```

The `or else` Operator

The `or else` operator is similar to the `and then` operator. In the following expression, the compiler evaluates `odd(x)` first, and if the result is true, does not evaluate `odd(y)`:

```
odd(x) or else odd(y)
```

Note – Comments cannot be inserted between the `or` and the `else` operators.

Example
The Pascal program, `or_else.p`. This program uses `or else` to test if two numbers are less than 10.

```
program or_else(input, output);

{ This program demonstrates the use
  of the operator or else. }

var
    x, y: integer16;

begin
    write('Please enter two integers:  ');
    readln(x, y);
    if (x < 10) or else (y < 10) then
      writeln('At least one number is less than 10.')
    else
      writeln('Both numbers are greater than or equal to 10.');
end. { or_else }
```

The commands to compile and execute `or_else.p`. This example shows the output when you input the numbers 101 and 3.

```
hostname% pc or_else.p
hostname% a.out
Please enter two integers: 101 3
At least one number is less than 10.
hostname%
```

Set Operators

The set operators in Table 4-7 accept different set types as long as the base types are compatible. The relational operators can also be used to compare set-type values.

Table 4-7 Set Operators

Operator	Operation	Operands	Result
+	set union	any set type	same as operands
-	set difference	any set type	same as operands
*	set intersection	any set type	same as operands
in	member of a specified set	2nd arg:any set type 1st arg:base type of 2nd arg	boolean

Relational Operators

The relational operators are given in Table 4-8. Note that in Pascal you can apply all relational operators to sets and the equality (=) and inequality (<>) operators on records and arrays.

Table 4-8 Relational Operators

Operator	Operation	Operand	Results
=	equal	any `real`, `integer`, `boolean`, `char`, record, array, set, or pointer type	`boolean`
<>	not equal	any `real`, `integer`, `boolean`, `char`, record, array, set, or pointer type	`boolean`
<	less than	any `real`, `integer`, `boolean`, `char`, string, or set type	`boolean`
<=	less than or equal	any `real`, `integer`, `boolean`, `char`, string, or set type	`boolean`
>	greater than	any `real`, `integer`, `boolean`, `char`, string, or set type	`boolean`
>=	greater than or equal	any `real`, `integer`, `boolean`, `char`, string, or set type	`boolean`

Relational Operators on Sets

Use the relational operators to compare sets of identical types. The result is a `boolean` (*true* / *false*) value.

Example
The Pascal program, `sets.p`, which applies the < and > operators to two sets of colors. The < operator tests if a set is a subset of another set. The > operator tests is a proper subset of another set.

```
program set_example(output);

{ This program demonstrates the use of relational
  operators on sets. }

var
    set1, set2: set of (red, orange, yellow, green);

begin
    set1 := [orange, yellow];
    set2 := [red, orange, yellow];
    writeln(set1 > set2);
    writeln(set1 < set2)
end. { set_example }
```

The commands to compile and execute `sets.p`:

```
hostname% pc sets.p
hostname% a.out
false
true
hostname%
```

The = and <> *Operators on Records and Arrays*

Use the = and <> operators to compare character arrays of the same size. For example, the following will apply:

- You can compare a `varying[10]` string with an `alfa` string.
- You cannot compare an `alfa` string with an `array[1..15]`.

In making comparisons, between arrays and records, make sure the operands are of the same type.

The Pascal program, `compare.p`, which makes comparisons among records.

```
program record_example(output);

const
    MAX = 10;

type
    Shape = (Square, Trapezoid, Rectangle);
    variant_record =
      record
          case Shape_type: Shape of
              Square: ( side1: real );
              Trapezoid: ( top1: real;
                           bottom: real;
                           height: real );
              Rectangle: ( length: real;
                           width: real )
      end;

    normal_record =
      record
          name: array [1..MAX] of char;
          avg: integer;
          grade: char
      end;

var
    class1: normal_record := ['Susan', 100];
    class2: normal_record := ['John', 99];
    shapes1: variant_record;
    shapes2: variant_record;
```

Comparing Records (Screen 1 of 2)

```
 begin
    { Should PASS. }
    if class1 <> class2 then
      writeln('PASSED')
    else
      writeln('FAIL');

    shapes1.Shape_type := Rectangle;
    shapes2.Shape_type := Square;
    { Should PASS }
    if shapes1 = shapes2 then
   writeln('FAIL')
    else
      writeln('PASSED');

    shapes1.Shape_type := Trapezoid;
    shapes2.Shape_type := Trapezoid;

    { Should PASS. }
    if shapes1 = shapes2 then
      writeln('PASSED')
    else
      writeln('FAIL')
end. { record_example }
```

Comparing Records (Screen 2 of 2)

The commands to compile and execute `compare.p`:

```
hostname% pc compare.p
hostname% a.out
PASSED
PASSED
PASSED
hostname%
```

String Operators

With the string concatenation operator, the plus sign (+), you can concatenate any combination of `varying`, `array of char`, constant strings, and single characters.

Example
The Pascal program, `concate.p`, which concatenates four types of strings.

```
program string_example(output);

{ This program demonstrates the use of
  the string concatenation operator. }

var
    col: varying [10] of char := 'yellow';
    fish: array [1..4] of char := 'tail';
    n1: char := 'o';
    n2: char := 'r';

begin
    write(fish + n1 + n2 + 'bird ', col + 'bird ');
    writeln(col + fish)
end. { string_example }
```

The commands to compile and execute `concate.p`:

```
hostname% pc concate.p
hostname% a.out
tailorbird yellowbird yellowtail
hostname%
```

Precedence of Operators

Table 4-9 lists the order of precedence of Pascal operators, from highest to lowest.

Table 4-9 Precedence of Operators

Operators	Precedence
`~, not,`	Highest
`*, /, div, mod, and, &,`	.
`|, !, +, -, or,`	.
`=, <>, <, <=, >, >=, in,`	.
`or else, and then`	Lowest

4

Program Heading and Declarations 5

This chapter describes Pascal program heading and declarations.

Program Heading

The syntax for the Pascal program heading follows:

```
<program heading> ::= 'program' <identifier>
    [ ' ( ' <identifier list> ' ) '  ] ' ; '
```

Pascal does not require you to declare `input` and `output` in the program heading. The file identifiers (other than `input` and `output`) must be declared as variables of type `file` in the global declaration part.

Declarations

The syntax for the Pascal declarations follows. Pascal allows you to specify these parts in any order and an unlimited number of times.

```
<declaration> ::= <label declaration>
    |  <constant declaration>
    |  <type declaration>
    |  <variable declaration>
    |  <define declaration>
    |  <procedure declaration>
    |  <function declaration>
```

The remaining sections of this chapter describe the label, constant, type, variable, and define declarations. Procedure and function declarations are described in the following chapter.

Label Declaration

The `label` declaration defines labels, which are used as the target of `goto` statements.

Syntax

```
<label declaration> ::= 'label' <label> {',' <label>} ';'
```

Comments

In Pascal, you can use both identifiers and integers as labels. Using identifiers as labels makes your code easier to read.

Example

The Pascal program, `label.p`:

```
program return_example;

{ This program demonstrates the use of the
  label declaration. }

var
    i: integer;

procedure test;
label
    error_negative_value, error_bad_values, error_value_too_big;
begin
    if i < 0 then
      goto error_negative_value
    else if (i = 2) or (i = 3) then
      goto error_bad_values
    else if i > 100 then
      goto error_value_too_big;
    return;
error_negative_value:
    writeln('Value of i must be greater than 0.');
    return;
error_bad_values:
    writeln('Illegal value of i:  2 or 3.');
    return;
error_value_too_big:
    writeln('Value of i too large.');
    return
end; { test }

begin { main procedure }
    write('Enter value for i:  ');
    readln(i);
    test
end. { return_example }
```

The commands to compile and execute `label.p`

```
hostname% pc label.p
hostname% a.out
Enter value for i: 101
Value of i too large.
hostname%
```

Constant Declaration

The constant declaration defines constants, values that do not change during program execution.

Syntax

```
<constant declaration> ::= 'const' <identifier> '='
      <constant expression> ';' { <identifier> '='
      <constant expression> ';' }
```

The value of *expression* may be a compile-time evaluable expression. It may contain any of the following:

- A `real`, `integer`, `boolean`, `character`, `set`, or `string` value.
- The pointer constant `nil`.
- Another previously defined constant.
- Predefined Pascal routines (see Chapter 7, “Input and Output) called with constant expression arguments if applicable.
- An operator (see Chapter 4, “Assignments and Operators).

Example

This constant declaration defines six valid constants.

```
const
    x = 75;
    y = 85;
    month = 'November';
    lie = false;
    result = (x + y) / 2.0;
    answer = succ(sqrt(5+4));
```

Type Declaration

The type declaration describes and names types used in variable, parameter, and function declarations.

Syntax

```
<type declaration> ::= 'type' <identifier> '=' <type> ';'
     { <identifier> '=' <type> ';' }
```

Unlike standard Pascal, Pascal allows you to define universal pointer types and procedure and function pointer types in the `type` declaration.

Example

This `type` declaration defines `opaque_pointers` as a universal pointer and `routines` as a function pointer.

```
type
    lowints = 0..100;
    primary_colors = (red, yellow, blue);
    opaque_pointers: univ_ptr;
    routines: function(i : integer): boolean;
    capital_letters = set of 'A'..'Z';
    digits = set of lowints;
    char_array = array[1..10] of char;
    record_type = record
       name: char_array;
       age : integer;
    end;
```

Variable Declaration

The variable declaration declares variables.

Syntax

```
<variable declaration> ::= [ <variable scope> ] 'var' <identifier list> ':'
     [ <variable attribute> ] <type specification> [ <initialization> ] ';'
     { <identifier list> ':' [ <variable attribute> ] <type specification>
     [ <initialization> ] ';' }
```

In the variable declaration, you can specify the variable scope, attributes, and initial values. In most cases, you will not have a variable declaration that has both a variable scope and a variable attribute because these are different ways for doing similar things.

Scope

The scope of a variable is either `private` or `public`.

- A `private` variable is visible in the current compilation unit only.
- A `public` variable is visible across multiple programs and modules.

The syntax for scope is as follows:

You may also use the `define/extern` declaration to declare a variable as public and the `static` attribute to declare a variable as private. See Chapter 9 for information on `define/extern`.

```
<variable scope> ::= ( 'private' | 'public' )
```

Variables in the `var` declaration section of a program default to `public` when you compile your program without the `–xl` option. When you compile your program with `–xl`, variables default to `private`.

This code declares both `public` and `private` variables.

```
public var
    total: single := 100.00;
    quantity: integer16 := 25;

private var
    score: integer16 := 99;
```

Attributes

The variable attributes determine how to allocate the variable and its scope. They include `static`, `extern`, and `define`. The syntax follows:

```
<variable attribute> ::= ( 'static' | 'extern' | 'define' )
```

`static`

A `static` variable is a variable that is private in scope and which is allocated statically. A global variable declared as `static` is equivalent to a variable that has been declared `private`. Pascal generates a compile-time error if you attempt to declare a global variable as both `static` and `public`.

When you declare a local variable as `static`, the variable retains its value after the program exits the procedure in which it is declared. You may only initialize a local variable, that is, a variable declared in a procedure, in the `var` declaration if you also declare it as `static`.

Example
The Pascal program, `static.p`:

```
program static_example;

{ This program demonstrates the use of the
  static variable attribute. }

var
    i: integer;

procedure count;

var
    number_of_times_called: static integer := 0;

begin
    number_of_times_called := number_of_times_called + 1;
   writeln('Call number: ', number_of_times_called)
end; { count }

begin { main program }
    for i := 1 to 4 do begin
      count
    end
end. { static_example }
```

The commands to compile and execute `static.p`:

```
hostname% pc static.p
hostname% a.out
Call number: 1
Call number: 2
Call number: 3
Call number: 4
hostname%
```

extern

The `extern` attribute is used to declare a variable that is not allocated in the current module or program unit, but is a reference to a variable allocated in another unit. You may not initialize `extern` variables. See the *Pascal 3.0.2 User's Guide*, which describes separately compiled programs and modules and contains examples of the `extern` attribute.

define

The `define` attribute is used to declare a variable that is allocated in the current module and whose scope is public. `define` is especially useful for declaring variables with the `-xl` option, which makes global variables private by default. See the *Pascal 3.0.2 User's Guide* for an example of this attribute.

Initialization

You can initialize `real`, `integer`, `boolean`, character, set, record, array, and pointer variables in the `var` declaration. You cannot initialize a local variable (a variable in the `var` declaration of a procedure or function) unless you declare it as `static`. The syntax for initialization is as follows:

```
<initialization> ::= ':='<initial values>
```

For details on the syntax for initialization see Appendix A, "Pascal Language Reference Summary

This example shows how to initialize a variable in the `var` declaration.

```
var
    x: array[1..5, 1..3] of real := [[* of 0.0], [* of 0.0]];
    year, zeta: integer := 0;
    sunny: boolean := false;
    c1: char := 'g';
    citrus: set of fruit := [orange, lemon, lime];
    name: array[1..11] of char := 'Rembrandt';
```

This code correctly declares the variables x, y, windy, and grade in procedure miscellaneous as static.

```
procedure miscellaneous;

var
    x: static integer16 := maxint;
    y: static single := 3.9;
    windy: static boolean := true;
    grade: static char := 'C';
```

Define Declaration

The `define` declaration controls the allocation of variables.

Syntax

```
<define declaration> ::= 'define' <identifier> [ <initialization> ]
    { ',' <identifier> [ <initialization> ] } ';'
```

Comments

The value of *identifier* must correspond to either a variable or procedure or function identifier. If *identifier* corresponds to a variable, it must have a matching variable declaration with the `extern` attribute. The `define` declaration nullifies the meaning of `extern`: it allocates the variable in the current program or module unit.

If *identifier* corresponds to a procedure or a function, it nullifies a previous `extern` procedure / function declaration; this means that you must define the procedure / function thereafter.

You may initialize variables, but not procedures and functions, in the `define` declaration. Identifiers in the `define` declaration are always `public`.

Example

See the chapter on separate compilation in the *Pascal 3.0.2 User's Guide* for examples of the `define` declaration.

Procedure and Function Declarations

<*procedure declaration*> ::= <*procedure heading*> <*declaration list*> <*block*>

<*function declaration*> ::= <*function heading*> <*declaration list*> <*block*>

Procedure and Function Headings

Here is the format of the procedure and function headings:

<*procedure heading*> ::= [<*visibility*>] '`procedure`' <*identifier*>
    ['(' <*parameter list*> ')'] ';' [<*routine options*>] ';'

<*function heading*> ::= [<*visibility*>] '`function`' <*identifier*>
    ['(' <*parameter list*> ')'] ':' <*type identifier*> ';'
    [<*routine options*>]

The following sections describe the visibility, parameter list, type identifier, and options.

Visibility

You can declare a procedure or function at the outer block level as either `public` or `private`.

When a procedure or function is public, you may reference that routine in another program or module unit. Declaring a routine as `private` restricts its accessibility to the current compilation unit. The syntax involved follows.

You may also use the `define/extern` declaration to declare a procedure or function as public and the `internal` routine option to declare a routine as private. For more information on the `define/extern` declaration, see Chapter 9.

<*visibility*> ::= (’`public`’ | ’`private`’)

Top-level procedures and functions declared in a program default to public when you compile your program without the `–xl` option. When you compile your program with `–xl`, all top-level routines declared in the program become private.

Nested procedures and functions are always private; it is illegal to declare a nested routine as public.

Procedures and functions declared within a module unit are always public. For additional information on modules, see the *Pascal 3.0.2 User’s Guide*.

This code fragment declares both public and private functions and procedures.

```
public procedure average(s,t: single);

private procedure evaluate(n : integer);

public function big (quart : integer16;
                cost : single) : single;

private function simple (x, y : boolean) : integer16;
```

Parameter List

Pascal supplies the parameter types `in`, `out`, `in out`, `var`, `value`, and `univ`. The syntax is given as:

<*parameter list*> ::= <*parameters*> { ';' <*parameters*> }

<*parameters*> ::= ([('in' | 'out' | 'in out' | 'var')]
    <*identifier list*> ':' ['univ'] <*type specification*>
    | 'procedure' <*identifier parameters*>
    | 'function' <*identifier parameters*> ':' <*type specification*>
    | <*conformant array parameters*>)

Parameters: `in`, `out`, *and* `in out`

The `in`, `out`, and `in out` parameters are extensions to the standard that allow you to specify the direction of parameter passing:

`in` indicates that the parameter can only pass a value into the routine. The parameter is, in effect, a "read-only" variable.

You may not assign a value to an `in` parameter. You may not pass an `in` parameter as an argument to another procedure that expects a `var`, `out`, or `in out` argument.

`out` indicates that the parameter will be used to pass values out of the routine. In effect, declaring a parameter as `out` tells the compiler that the parameter has no initial value and that assignments to the parameter will be retained by the caller.

`in out` indicates that the parameter can both take in values and pass them back out. An `in out` parameter is equivalent to a `var` parameter.

Example

The Pascal program, `in_out.p`. The procedure `compute_area` reads in the `length` and `width` and outputs `result`. The procedure `multiply_by_two` reads in `result`, multiplies it by two and returns the modified value.

```
program in_out_example(input, output);

{ This program, which finds the area of a rectangle,
  demonstrates the use of the in, out, and in out
  parameters. }

var
    length, width, result: real;

{ Find area given length and width. }
procedure compute_area(in length: real; in width: real;
                       out result: real);

begin
    result := length * width
end; { compute_area } { compute_area }


{ Multiply the area by two. }
procedure multiply_by_two(in out result: real);

begin
    result := result * 2
end; { multiply_by_two } { multiply_by_two }


begin { main program }
    write('Enter values for length and width:  ');
    readln(length, width);
    compute_area(length, width, result);
    writeln('The area is ', result: 5: 2, '.');
    multiply_by_two(result);
    writeln('Twice the area is ', result: 5: 2, '.')
end. { in_out_example }
```

The commands to compile and execute `in_out.p`. This example shows the program output when you input a length of 4 and a width of 5.

```
hostname% pc in_out.p
hostname% a.out
Enter values for length and width: 4 5
The area is 20.00.
Twice the area is 40.00.
hostname%
```

`var` *Parameters*

`var` parameters are the same in standard Pascal and Pascal.

Value Parameters

Value parameters are the same in standard Pascal and Pascal.

`univ` *Parameters*

The nonstandard `univ` parameter type is actually a modifier used before data types in formal parameter lists to turn off type-checking for that parameter. You can use it with any type of parameter except conformant array, procedure, or function parameters.

`univ` is used as follows:

```
procedure somename (var firstparam: univ integer);
```

You could then call this procedure with a parameter of any type. You should always declare a `univ` parameter as either `in`, `out`, `in out`, or `var`.

`univ` is most often used for passing arrays, where you may call a procedure or function with different array sizes. In that case, you generally would pass another parameter that gave the actual size of the array as follows:

```
type
    real_array = array[1..100] of real;

procedure receive(size: integer;
            var theArray: univ real_array);

var
    n: integer;

begin
     for n:= 1 to size do
       .
       .
       .
```

Type Identifier

In Pascal, a function may represent a structure, such as a set, array, or record. In standard Pascal, a function may only represent the simple types of value, ordinal, or real.

Functions Returning Strings and Varying Strings

When declaring functions that return strings (arrays of chars) and varying strings, you can specify the result by an assignment. For example:

```
F:= 'The answer: 12 miles'
```

where F is the function. However, sometimes you may want to obtain the string result by modifying some of the characters of an existing string (variable or parameter). In the example below, you may want to substitute a string for the string 'XX':

```
program String_Function_Example;
type s1 = array [1..20] of char;
    s2 = array [1..2 ] of char;

function f(x:s2):s1;
begin
  f := 'The answer: XX miles';
  f[13]:=x[1];
  f[14]:=x[2];
end;

var r: s2;
    s: s1;
begin
   r:='12';
   s:=f(r);

   writeln(s)
end.
```

In general, an identifier of a function `f` returning a string can be used in an assignment of the kind:

```
f[i]:=c
```

for specifying the i'th byte of the function result. This Pascal extension can be used both for strings and varying strings.

Options

Pascal supplies the standard `forward` routine option and the nonstandard options `extern`, `external`, `internal`, `variable`, and `nonpascal`. The syntax involved is:

```
<routine options> ::= 'options' '(' <options> { ',' <options> } ')'
     | { <options> ';' } ')'
```

```
<options> ::= ('forward' | 'extern'['('fortran' | 'c')]
    | 'external'  [ ( 'fortran' | 'c' ) ]
    | 'internal' | 'variable' |  'nonpascal'
```

forward

The `forward` option is the same in Pascal and standard Pascal.

extern *and* external

The `extern` and `external` options indicate that the procedure or function is defined in a separate program or module. `extern` and `external` allow the optional specification of the source language of the procedure or function. For more information on these options, see the chapter on separate compilation in the *Pascal 3.0.2 User's Guide*.

internal

The `internal` option makes the procedure or function local to that module. Specifying the `internal` option is the same as declaring the procedure or function as `private`. Pascal generates an error message if you attempt to declare a `public` procedure or function as `internal`.

variable

The `variable` option allows you to pass a procedure or function a smaller number of actual arguments than the number of formal arguments defined in the routine. The actual arguments must match the formal parameters types. You may not pass a larger number of actual arguments than formal arguments.

Example

The Pascal program, `variable.p`, passes either two or three actual arguments to the procedure `calculate_total`, depending on the user input.

```
program variable_example(input, output);

{ This program demonstrates the use of the
  variable routine option. }

const
    tax_rate = 0.07;
    shipping_fee = 2.50;

var
    price: single;
    resident: char;
    total: single;

function calculate(count: integer16; price: single;
                   tax: single): single;
                   options(variable);

begin
    if count = 2 then
      calculate := price + tax + shipping_fee
    else
      calculate := price + shipping_fee
end; { calculate }

begin { main program }
    write('Please enter the price:  ');
    readln(price);
     writeln('California residents must add local sales tax.');
     write('Are you a California resident?  Enter y or n:  ');
    readln(resident);
    if resident = 'y' then
      total := calculate(2, price, tax_rate * price)
    else
      total := calculate(1, price);
    writeln('Your purchase amounts to $', total: 5: 2, '.')
end. { variable_example }
```

The commands to compile and execute `variable.p`:

```
hostname% pc variable.p
hostname% a.out
Please enter the price: 10.00
California residents must add local sales tax.
Are you a California resident? Enter y or n: y
Your purchase amounts to $13.20.
hostname% a.out
Please enter the price: 10.00
California residents must add local sales tax.
Are you a California resident? Enter y or n: n
Your purchase amounts to $12.50.
hostname%
```

nonpascal

Pascal supports `nonpascal` as a routine option when you compile your program with the `-xl` option. `nonpascal` declares non-Pascal routines when you are porting Apollo DOMAIN programs written in DOMAIN Pascal, FORTRAN or C.

`nonpascal` passes arguments by reference. If the argument is a variable, `nonpascal` passes its address. If the argument is a constant or expression, `nonpascal` makes a copy on the caller's stack and passes the address of the copy.

Built-in Procedures and Functions 6

This chapter describes the built-in procedures and functions Pascal supports. The chapter is divided into three sections. The first section lists the standard routines Pascal supplies. The second section summarizes the nonstandard Pascal routines by the function they perform. The third section lists the nonstandard routines alphabetically and contains detailed descriptions and examples of each routine.

Standard Procedures and Functions

Pascal supplies the standard procedures in Table 6-1 and the standard functions in Table 6-2.

Table 6-1 Standard Procedures

`dispose`	`page`	`reset`	`writeln`
`get`	`put`	`rewrite`	
`new`	`read`	`unpack`	
`pack`	`readln`	`write`	

6

Table 6-2 Standard Functions

`abs`	`eof`	`odd`	`sin`	`trunc`
`arctan`	`eoln`	`ord`	`sqr`	
`chr`	`exp`	`pred`	`sqrt`	
`cos`	`ln`	`round`	`succ`	

Routines Specific to Pascal

This sections lists the nonstandard Pascal procedures and functions according to the following categories:

- Arithmetic routines
- Bit-shift routines
- Character string routines
- Input and output routines
- Miscellaneous routines

Table 6-3 Non Standard Arithmetic Routines

Routine	**Description**
`addr`	Returns the address of a variable, constant, function, or procedure.
`card`	Returns the cardinality of a set.
`expo`	Calculates the exponent of a variable.
`firstof`	Returns the first possible value of a type or variable.
`in_range`	Determines whether a value is in the defined integer subrange.
`lastof`	Returns the last possible value of a type or variable.
`max`	Returns the larger of two expressions.
`min`	Returns the smaller of two expressions.

Table 6-3 Non Standard Arithmetic Routines

Routine	Description
`random`	Generates a random number between 0.0 and 1.0.
`seed`	Resets the random number generator.
`sizeof`	Returns the size of a designated type or variable.

Table 6-4 Non Standard Bit Shift Routines

Routine	Description
`arshft`	Does an arithmetic right shift of an integer.
`asl`	Does an arithmetic left shift of an integer.
`asr`	Identical to `arshft`.
`land`	Returns the bitwise `and` of two integers.
`lnot`	Returns the bitwise `not` of an integer.
`lor`	Returns the inclusive `or` of two integers.
`lshft`	Does a logical left shift of an integer.
`lsl`	Identical to `lshft`.
`lsr`	Identical to `rshft`.
`rshft`	Does a logical right shift of an integer.
`xor`	Returns the exclusive `or` of two integers.

Table 6-5 Non Standard Character String Routines

Routine	Description
`index`	Returns the position of the first occurrence of a string or character inside another string.
`length`	Returns the length of a string.
`substr`	Extracts a substring from a string.
`trim`	Removes all trailing blanks in a character string.

Table 6-6 Non Standard Input and Output Routines

Routine	Description
`close`	Closes a file.
`flush`	Writes the output buffered for a Pascal file into the associated operating system file.
`getfile`	Returns a pointer to the C standard I/O descriptor associated with a Pascal file.
`linelimit`	Terminates program execution after a specified number of lines has been written into a text file.
`message`	Writes the specified information on `stderr`.
`open`	Associates an external file with a file variable.
`remove`	Removes the specified file.

Table 6-7 Extensions to Standard Input and Output Routines

Routine	Description
`read` and `readln`	Reads in boolean variables, fixed- and variable-length strings, and enumerated types from the standard input.
`reset` and `rewrite`	Accept an optional second argument, an operating system file name.
`write` and `writeln`	Outputs enumerated type values to the standard output.
	Outputs expressions in octal or hexadecimal.
	Allows negative field widths.

Table 6-8 Miscellaneous Nonstandard Routines

Routine	Description
`argc`	Returns the number of arguments passed to the program.
`argv`	Assigns the specified program arguments a string variable.
`clock`	Returns the user time consumed by this process.
`date`	Fetches the current date.
`discard`	Explicitly discards the return value of a function
`getenv`	Returns the value associated with an environment name.
`halt`	Terminates program execution.
`null`	Performs no operation.
`pcexit`	Terminates the program and returns an exit code.
`stlimit`	Terminates program execution if a specified number of statements have been executed in the current loop
`sysclock`	Returns the system time consumed by this process.
`time`	Retrieves the current time.
type_transfer	Permits you to change the data type of a variable or expression
`wallclock`	Returns the elapsed number of seconds since 00:00:00 GMT January 1, 1970.

addr

The `addr` function returns the address of a variable, constant, function, or procedure.

Syntax

`addr`(*x*)

Arguments

x is either a variable, a constant string, a function, or a procedure.

Return Value

The return value of `addr` is the address in which the variable or a constant string is stored. For function or procedural arguments, `addr` returns the starting address of the function or procedure. In each case, `addr` returns a value of type `univ_ptr`.

Comments

Pascal allows you to apply `addr` to a variable, function, or procedure with dynamic extent such as local variables and nested functions or procedures. You must exercise caution in doing so and then dereferencing the resulting pointer value. In the case of local variables, dereferencing these pointers outside the scope in which the variable is active will result in a meaningless value.

The compiler passes a static link to nested functions and procedures when it calls them. The compiler does not generate this link when dereferencing pointer values to procedures or functions. Consequently, Pascal generates a warning if the argument to `addr` is any of these objects.

`addr` cannot be applied to bit-aligned fields of aggregates.

Note – If you use the `addr ( )` function, don't use the `–H` option. The `–H` option makes sure that all pointers used point into the heap.

Example

The Pascal program, `addr.p`:

```
program addr_example(output);

{ This program demonstrates the use of the
  addr function. }

const
    name = 'Gail';

type
    ptr = ^ integer;
    ptr_char = ^ alfa;

var
    ptr_address: ptr;
    ptr_address_char: ptr_char;
    x: integer;
    y: integer;
    c: alfa;

begin
    x := maxint;

    { Get the address of x. }
    ptr_address := addr(x);

    { Get the contents of ptr_address. }
    y := ptr_address^;
    writeln('The address of x is  ', ptr_address: 3, '.');
    writeln('The contents of x is ', y: 3, '.');

    { Get the address of the constant name. }
    ptr_address_char := addr(name);

    { Get the contents of ptr_address_char. }
    c := ptr_address_char^;

    writeln('The address of c is  ', ptr_address_char: 3, '.');
    writeln('The contents of c is ', c: 4, '.')
end. { addr_example }
```

The commands to compile and execute `addr.p`:

```
hostname% pc addr.p
hostname% a.out
The address of x is 38764.
The contents of x is 2147483647.
The address of c is 33060.
The contents of c is Gail.
hostname%
```

argc

The `argc` function returns the number of arguments passed to the program.

Syntax

`argc`

Arguments

`argc` does not take any arguments.

Return Value

`argc` returns an integer value.

Comments

The return value of `argc` is always at least 1, the name of the program.

`argc` is normally used in conjunction with the built-in procedure `argv`. See the `argv` listing.

Example

See the example in the `argv` listing.

argv

The `argv` procedure assigns the specified program argument to a string variable.

Syntax

`argv(`*i*,*a*`)`

Arguments

i is an integer value.

a is a fixed- or variable-length string.

Return Value

`argv` returns a string variable.

Comments

`argv` returns the *i*'th argument of the current process to the string variable *a*. *i* ranges from 0, the program name, to `argc`-1.

`argc` is a predeclared function that tells you how many arguments are being passed to the program. `argv` is normally used in conjunction with `argc`.

Example

The Pascal program, `argv.p`:

```
program argv_example(output);

{ This program demonstrates the use of
  argc and argv. }

var
    i: integer32;
    name: varying [30] of char;

begin
    { Argument number 0 is the name of the program. }
    argv(0, name);
    writeln('The name of the program is ', name, '.');
    i := 1;
    while i <= argc - 1 do begin
      argv(i, name);
      writeln('Argument number ', i: 1, ' is ', name, '.');
      i := i + 1
    end
end. { argv_example }
```

The commands to output and execute `argv.p`:

```
hostname% pc argv.p
hostname% a.out
The name of the program is a.out.
hostname% a.out one two three
The name of the program is a.out.
Argument number 1 is one.
Argument number 2 is two.
Argument number 3 is three.
hostname%
```

arshft

The `arshft` function does an arithmetic right shift of an integer value.

Syntax

`arshft(`*num*,*sh*`)`

Arguments

num and *sh* are integer expressions.

Return Value

`arshft` returns a 32-bit integer value.

Comments

`arshft` shifts the bits in *num sh* places to the right. `arshft` preserves the sign bit of *num*. `arshft` does not wrap bits around from left to right. The sign bit is the most significant (left-most) bit in the number. Pascal uses two's complement to represent negative integers. For example, -8 as a 16-bit integer is represented as:

```
1111 1111 1111 1000
```

If you shift this number to the right by 1:

```
(arshft (-8, 1) )
```

your result will be:

```
1111 1111 1111 11er100
```

However, if you shift to the right by 1,

```
0111 1111 1111 1000
```

you result will be:

```
0011 1111 1111 1100
```

The `arshft` returns result is machine-dependent and is unspecified unless the following is true:

```
0 <= sh <= 32
```

Example

The Pascal program, `arshft.p`:

```
program arshft_example(input, output);

{ This program demonstrates the arithmetic right shift. }

const
    SIZE = 8;

var
    i: integer32;
    i32: integer32;
    loop: integer32;

begin
    write('Enter a positive or negative integer:  ');
    readln(i);
    for loop := 1 to SIZE do begin
      i32 := arshft(i, loop);
      write('Arithmetic right shift ', loop: 2);
      writeln(' bit(s):  ', i32 hex)
    end
end. { arshft_example }
```

The commands to compile and execute `arshft.p`. The value the bit-shift routines return may depend upon the architecture of your system.

```
hostname% pc arshft.p
hostname% a.out
Enter a positive or negative integer: -2
Arithmetic right shift 1 bit(s): FFFFFFFF
Arithmetic right shift 2 bit(s): FFFFFFFF
Arithmetic right shift 3 bit(s): FFFFFFFF
Arithmetic right shift 4 bit(s): FFFFFFFF
Arithmetic right shift 5 bit(s): FFFFFFFF
Arithmetic right shift 6 bit(s): FFFFFFFF
Arithmetic right shift 7 bit(s): FFFFFFFF
Arithmetic right shift 8 bit(s): FFFFFFFF
hostname%
```

asl

The `asl` function does an arithmetic left shift of an integer value.

Syntax

`asl(`*num*, *sh*`)`

Arguments

num and *sh* are integer expressions.

Return Value

`asl` returns a 32-bit integer value.

Comments

`asl` shifts the bits in *num sh* places to the left. `asl` preserves the sign bit of *num*. `asl` does not wrap bits from left to right.

The result `asl` returns is machine-dependent and is unspecified unless the following is true:

```
0 <= sh <= 32
```

Example

The Pascal program, `asl.p`:

```
program asl_example(input, output);

{ This program demonstrates the arithmetic left shift. }

const
    SIZE = 8;

var
    i: integer32;
    i32: integer32;
    loop: integer32;

begin
    write('Enter a positive or negative integer:  ');
    readln(i);
    for loop := 1 to SIZE do begin
      i32 := asl(i, loop);
      write('Arithmetic left shift ', loop: 2);
      writeln(' bit(s):  ', i32 hex)
    end
end. { asl_example }
```

The commands to compile and execute `asl.p`.

```
hostname% pc asl.p
hostname% a.out
Enter a positive or negative integer: 19
Arithmetic left shift 1 bit(s): 26
Arithmetic left shift 2 bit(s): 4C
Arithmetic left shift 3 bit(s): 98
Arithmetic left shift 4 bit(s): 130
Arithmetic left shift 5 bit(s): 260
Arithmetic left shift 6 bit(s): 4C0
Arithmetic left shift 7 bit(s): 980
Arithmetic left shift 8 bit(s): 1300
hostname%
```

asr

The `asr` function is identical to the `arshft` function. See the `arshft` listing earlier in this chapter.

card

The `card` function returns the number of elements in a set variable.

Syntax

`card(`*x*`)`

Arguments

x must be a set variable.

Return Value

`card` returns an integer value.

Comments

`card` returns the number of elements in the actual set variable, not the size of the set type.

Example

The Pascal program, `card.p`:

```
program card_example(output);

{ This program demonstrates the use of the card function. }
type
    lowints = 0..100;
    primary_colors = set of (red, yellow, blue);
    possibilities = set of boolean;
    capital_letters = set of 'A'..'Z';
    digits = set of lowints;

var
    pri: primary_colors;
    pos: possibilities;
    cap: capital_letters;
    dig: digits;

begin
    pri := [red, yellow, blue];
    pos := [true, false];
    cap := ['A'..'Z'];
    dig := [0..100];
    writeln('There are ',card(pri): 4, ' primary colors.');
    writeln('There are ',card(pos): 4, ' possibilities.');
    writeln('There are ',card(cap): 4, ' capital letters.');
    writeln('There are ',card(dig): 4, ' digits.')
end. { card_example }
```

The commands to output and execute `card.p`:

```
hostname% pc card.p
hostname% a.out
There are     3 primary colors.
There are     2 possibilities.
There are    26 capital letters.
There are   101 digits.
hostname%
```

clock

The `clock` function returns the user time consumed by the process.

Syntax

```
clock
```

Arguments

`clock` does not take any arguments.

Return Value

`clock` returns an integer value.

Comments

`clock` returns the user time in milliseconds.

See also the `sysclock` function, which returns the system time the process uses.

Example

The Pascal program, `clock.p`:

```
program clock_example(input, output);

{ This program times how long it takes to run the
  towers of hanoi. }

const
    DISK = 16;

var
    num: array [1..3] of integer;
    counts: integer32;
    before_user: integer;
    before_sys: integer;
    after_user: integer;
    after_sys: integer;

procedure moves(number, f, t: integer);

var
    o: integer;

begin
    if number = 1 then begin
      num[f] := num[f] - 1;
      num[t] := num[t] - 1;
      counts := counts + 1
    end else begin
      o := 6 - (f + t);
      moves(number - 1, f, o);
      moves(1, f, t);
      moves(number - 1, o, t)
    end
end; { moves } { moves }
```

`clock.p` program (Screen 1 of 2)

```
begin { main program }
    before_user := clock;
    before_sys := sysclock;
    moves(DISK, 1, 3);
    after_sys := sysclock;
    after_user := clock;
    write('For ', DISK: 1, ' disks, there were ');
    writeln(counts: 1, ' steps.');
    write('Elapsed system time:  ');
    writeln(after_sys - before_sys: 1, ' milliseconds.');
    write('Elapsed user time:  ');
    writeln(after_user - before_user: 1, ' milliseconds.')
end. { clock_example }
```

`clock.p` program (Screen 2 of 2)

The commands to compile and execute `clock.p`. The time `clock` and `sysclock` return is system-dependent.

```
hostname% a.out
For 16 disks, there were 65535 steps.
Elapsed system time: 16 milliseconds.
Elapsed user time: 583 milliseconds.
hostname%
```

close

The `close` procedure closes a file.

Syntax

`close(`*file*`)`

Arguments

file is a file having the `text` or `file` data type.

Return Value

`close` does not return any values.

Comments

`close` closes the open file named *file*. `close` is optional; Pascal closes all files either when the program terminates or the program leaves the procedure in which the file variable is associated with the open file.

Pascal generates a runtime error if *file* is not an open file. You may trap this error with the I/O error recovery mechanism, described in *Section , "I/O Error Recovery* in Chapter 7, "Input and Output

Pascal will not allow you to close the predeclared files `input` and `output`. If you redirect `input` or `output`, the associated streams are automatically closed.

See also the `open`, `reset`, and `rewrite` procedures, which open a file.

Example

See the example in the `open` listing in this chapter.

date

The `date` procedure fetches the current date (as assigned when the operating system was initialized) and assigns it to a string variable.

Syntax

`date`(*a*)

Arguments

a is a variable that can be either a character array 10 elements long or a variable-length string.

Return Value

`date` returns a character string in the form `dd mmm yy`, where `dd` is the day, `mmm` is the first three characters of the month, and `yy` is the year.

Comments

`date` puts a zero in front of the day and year (07) to make them have two digits. Also, `date` stores the answer with the day, month, and year separated by spaces. `date` puts a blank at the end of the string.

Example

The Pascal program, `date.p`:

```
program date_example(output);

{ This program demonstrates the date procedure. }

var
    s1: alfa;
    s2: array [1..10] of char;
    s3: array [89..98] of char;
    s4: varying [100] of char;

begin
    date(s1);
    date(s2);
    date(s3);
    date(s4);
    writeln('The date is ', s1, '.');
    writeln('The date is ', s2, '.');
    writeln('The date is ', s3, '.');
    writeln('The date is ', s4, '.')
end. { date_example }
```

The commands to compile and execute `date.p`:

```
hostname% pc date.p
hostname% a.out
The date is 23 May 89 .
The date is 23 May 89 .
The date is 23 May 89 .
The date is 23 May 89 .
hostname%
```

discard

The `discard` procedure throws away the value of an expression.

Syntax

`discard(`*expr*`)`

Arguments

expr is any expression including a function call.

Return Value

`discard` does not return any values.

Comments

Use `discard` to call a function or evaluate an expression whose value you do not need to continue program execution. For example, you can use discard to execute a function whose return value you do not need.

Example

The Pascal program, `discard.p`:

```
program discard_example(output);

{ This program computes a discount if the total amount
  is over DISC_AMOUNT. }

const
    RATE = 0.15;
    DISC_AMOUNT = 100.00;

var
    amount: single;
    discount: single;

function compute(amount: single): single;

begin
    compute := amount * RATE
end; { compute }

begin { main program }
    write('Enter sale amount:  ');
    readln(amount);
    if amount < DISC_AMOUNT then begin
      discard(compute(amount));
      write('No discount applied; total charge amount');
    writeln(' must be more than ', DISC_AMOUNT:2: 2, '.')
    end else begin
      discount := compute(amount);
      write('The amount of discount on ');
      writeln(amount: 2: 2, ' is ', discount: 2: 2, '.')
    end
end. { discard_example }
```

The commands to compile and execute `discard.p`:

```
hostname% pc discard.p
hostname% a.out
Enter sale amount: 125.00
The amount of discount on 125.00 is 18.75.
hostname%
```

6

expo

The expo function calculates the integer-valued exponent of a specified number.

Syntax

expo(*x*)

Arguments

x is either a real or integer value.

Return Value

expo returns an integer value.

Comments

expo returns an integer that represents the integer-valued exponent of a real number.

Example

The Pascal program, `expo.p`:

```
program expo_example(output);

{ This program demonstrates the expo function. }

const
    MAX = 10;

var
    i: integer;
    r: real;

begin
    writeln(' x   r := exp(x)             expo(r)');
    writeln(' -   -----------             -------');
    for i := 1 to MAX do begin
      r := exp(i);
      writeln(i: 2, '  ', r, '  ', expo(r))
    end
end. { expo_example }
```

The value `expo` returns may depend upon the architecture of your system.

```
hostname% pc expo.p
hostname% a.out
 x r := exp(x)        expo(r)
 - ----------- -------
 1 2.71828182845905e+00   0
 2 7.38905609893065e+00   0
 3 2.00855369231877e+01   1
 4 5.45981500331442e+01   1
 5 1.48413159102577e+02   2
 6 4.03428793492735e+02   2
 7 1.09663315842846e+03   3
 8 2.98095798704173e+03   3
 9 8.10308392757538e+03   3
10 2.20264657948067e+04   4
hostname%
```

6

firstof

The `firstof` function returns the value of the lower bound when its argument is or has an ordinal type. For array types, `firstof` returns the lower bound for the subrange defining the array index. For set types, it returns the lower bound of the set base type.

Syntax

`firstof`(*x*)

Arguments

x is either a variable, a constant, an expression, or the name of a user-defined or predeclared Pascal data type. *x* may not be a record, a file, a pointer type, a conformant array, a procedure or function parameter, or a string literal.

Return Value

When *x* is an ordinal type, a constant, an expression, or variable, the value `firstof` returns has the same data type as its argument.

When *x* is an array, the value `firstof` returns has the same data type as the type of the array index.

When *x* is a set type, the value `firstof` returns has the same data type as the base type of the set.

Comments

Pascal follows the rules in Table 6-9 when returning the value of *x*.

Table 6-9 `firstof` Return Values

Type of Argument	Return Value
`integer` (without `–xl` option)	-2147483648
`integer` (with `–xl` option)	-32768
`integer16`	-32768
`integer32`	-2147483648
`char`	chr(0)
`boolean`	false
`enumerated`	The first element in the enumeration type declaration.
`array`	The lower bound of the subrange hat defines the array size.
`varying`	1
`set of 'A'..'Z'`	'A' (the character 'A').

Example

See the examples that follow.

The Pascal program, `firstof.p`:

```
program firstof_example(output);

{  This program illustrates the use of firstof and lastof
   used with arrays and enumerated types. }

const
    dollars_per_tourist = 100;

type
     continents = (North_America, South_America, Asia, Europe, Africa, Australia,
    Antarctica);

var
    i: continents;
    major_targets: array [continents] of integer := [20, 3, 15, 25, 5, 1, 0];
    planned_targets: array [continents] of integer := [* of 0];

begin
    for i := firstof(planned_targets) to lastof(planned_targets) do begin
      planned_targets[i] := major_targets[i] * dollars_per_tourist
    end;

    for i := firstof(continents) to lastof(continents) do begin
        writeln(i, ' is the goal of ', planned_targets[i]: 1, ' dollars per tourist.')
    end

end. { firstof_example }
```

The commands to compile and execute `firstof.p`:

```
hostname% pc firstof.p
hostname% a.out
North_America is the goal of 2000 dollars per tourist.
South_America is the goal of 300 dollars per tourist.
Asia is the goal of 1500 dollars per tourist.
Europe is the goal of 2500 dollars per tourist.
Africa is the goal of 500 dollars per tourist.
Australia is the goal of 100 dollars per tourist.
Antarctica is the goal of 0 dollars per tourist.
hostname%
```

flush

The `flush` procedure writes the output buffer for the specified Pascal file into the associated file.

Syntax

`flush(`*file*`)`

Arguments

file is a file having the `text` or `file` data type.

Return Value

`flush` does not return any values.

Comments

The `flush` procedure causes the compiler to write all characters buffered for output to the specified file.

For example, in the following code fragment the compiler writes the output integer *i* to the file *f* when it encounters `flush`:

```
for i := 1 to 5 do begin
    write(f,i);
    Compute a lot with no output
 end;
flush(f);
```

`flush` does not append a newline character after writing the data. See also the output procedures `message`, `write`, and `writeln`.

Example

The Pascal program, `flush.p`:

```
program flush_example(output);

{ This program demonstrates the use of the
  flush procedure. }

const
    NAME = 'flush.txt';
var
    i: integer;
    f1, f2: text;

procedure read_file;
var
    i: integer;
begin
    reset(f2, NAME);
    writeln('Beginning of file.');
    while not eof(f2) do begin
      while not eoln(f2) do begin
          read(f2, i);
          writeln(i)
      end;
      readln(f2)
    end;
    writeln('End of file.');
    writeln
end; { read_file }
```

`flush.p` (Screen 1 of 2)

```
begin { main program }
    rewrite(f1, NAME);
    for i := 1 to 10 do
      write(f1, i);

    { At this point the file is still empty. }
    read_file;

    flush(f1);

    { Now the file contains data after the flush. }
    read_file
end. { flush_example }
```

`flush.p` (Screen 2 of 2)

The commands to compile and execute `flush.p`:

```
hostname% pc flush.p
hostname% a.out
Beginning of file.
End of file.

Beginning of file.
     1
     2
     3
     4
     5
     6
     7
     8
     9
    10
End of file.

hostname%
```

6

getenv

The `getenv` function returns the value associated with an environment name.

Syntax

`getenv(`*string, string_variable*)

Arguments

string is either a constant string, a variable-length string, or a character array. *string_variable* is a variable-length string or a character array.

Return Value

`getenv` returns a variable-length string or a character array.

Comments

The variable *string* is an environment name. Pascal returns the value for the environment name through the parameter *string_variable*.

string must match the environment exactly, and trailing blanks are significant. If *string* is a character array, you may want to use the `trim` function.

If there are no environment names with the value *string*, the value of *string_variable* is the null string if *string_variable* is a variable-length string. If *string_variable* is a character array, it is padded with blanks.

See the *SunOS Reference Manual* (for SunOS 5.0) for a complete description of environment variables.

Example

The Pascal program, `getenv.p`:

```
program getenv_example;

{ This program demonstrates the use of the
  getenv function. }

var
    namev: varying [10] of char := 'EDITOR';
    names: array [1..10] of char := 'EDITOR';
    valv: varying [20] of char;

begin
    getenv(namev, valv);
    writeln(namev, ' = ', valv);
    getenv(trim(names), valv);
    writeln(names, ' = ', valv)
end. { getenv_example }
```

The commands to compile and execute `getenv.p`:

```
hostname% pc getenv.p
hostname% a.out
EDITOR = /usr/ucb/vi
EDITOR = /usr/ucb/vi
```

getfile

The `getfile` function returns a pointer to the C standard I/O descriptor associated with a Pascal file.

Syntax

`getfile(`*file*`)`

Arguments

file is a file having the `text` or `file` data type. *file* must be associated with an open file; otherwise `getfile` returns `nil`.

Return Value

`getfile` returns a value of type `univ_ptr`.

Comments

You may use the result of `getfile` for files opened with either the `reset`, `rewrite`, or `open` procedures, placing the return value as a parameter to a C I/O routine. Use extreme caution when you call `getfile`; directly calling C I/O routines circumvents bookkeeping data-structures in the Pascal I/O library.

As a general rule, calling C routines for writing is safe. Using the return value for calling C routines for reading may cause subsequent `eoln`, `eof`, or `readln` calls to misbehave for that file.

Example

The Pascal program, `getfile.p`:

```
program getfile_example;

{ This program demonstrates the use of the
  getfile function. }

type
    char_array = array [1..30] of char;

var
    f: text;
    cfile: univ_ptr;

procedure fprintf(cf: univ_ptr; in format: char_array;
                  in year: integer); external c;

begin { main program }
    rewrite(f, 'output.data');
    cfile := getfile(f);
    fprintf(cfile, 'Hello, world, in the year %d .', 1990)
end. { getfile_example }
```

The commands to compile and execute `getfile.p`:

```
hostname% pc getfile.p
hostname% a.out
hostname% more output.data
Hello, world, in the year 1990 .
hostname%
```

halt

The `halt` procedure terminates program execution.

Syntax

`halt`

Arguments

`halt` does not take any arguments.

Return Values

`halt` does not return any values.

Comments

You may use `halt` anywhere in a program to terminate execution. When execution of a program encounters a `halt`, it prints the following message:

```
Call to procedure halt
```

Pascal returns to command level after it executes a `halt`.

Example

The Pascal program, `halt.p`:

```
program halt_example(input, output);

{ This program calculates a factorial. }

var
    x, y: integer;

function factorial(n: integer): integer;

begin
    if n = 0 then
      factorial := 1
    else
      factorial := n * factorial(n - 1)
end; { factorial } { factorial }

begin { main program }
    write('Enter a positive integer from 0 to 16:  ');
    readln(x);
    if (x >= 0) and (x <= 16) then begin
      y := factorial(x);
      write('The factorial of ', x: 1);
      writeln(' is ', y: 1, '.')
    end else begin
      writeln('Illegal input.');
      halt
    end
end. { halt_example }
```

The commands to compile and execute `halt.p`:

```
hostname% pc halt.p
hostname% a.out
Enter a positive integer from 0 to 16: 8
The factorial of 8 is 40320.
hostname% a.out
Enter a positive integer from 0 to 16: 20
Illegal input.
Call to procedure halt
hostname%
```

in_range

The `in_range` function checks if a value is in the defined subrange.

Syntax

`in_range(`*x*`)`

Arguments

x is an integer, `boolean`, character, enumerated, or subrange data type.

Return Value

`in_range` returns a `boolean` value.

Comments

`in_range` returns `true` if *x* is in the defined range or `false` if *x* is outside the range.

`in_range` is useful for doing a runtime check to see if *x* has a valid value. `in_range` is especially helpful for checking enumerated and subrange types. However, this feature does not work for 32-bit integer values.

If you compile your program with the **`-C`** option, the compiler also generates code that does range checking. However, if the variable is out of range, the program terminates. By using `in_range` instead, you can control subsequent execution of your program.

Example

The Pascal program, `in_range.p`:

```
program in_range_example(input, output);

{ This program demonstrates the use of the in_range function. }

type
    positive = 1..maxint;

var
    base, height: positive;
    area: real;

begin
    write('Enter values for triangle base and height:  ');
   readln(base, height);
    if in_range(base) and in_range(height) then begin
      area := base * height / 2;
      writeln('Area is ', area: 5: 2, '.')
   end else
      writeln('Cannot compute negative areas.')
end. { in_range_example }
```

The commands to compile and execute `in_range.p`:

```
hostname% pc in_range.p
hostname% a.out
Enter values for triangle base and height: 4 5
Area is 10.00.
hostname%
```

index

The `index` function returns the position of the first occurrence of a string or character within another string.

Syntax

`index(`*target_string*, *pattern_string*`)`

Arguments

target_string is a constant string, variable-length string, or an array of character.

pattern_string is a constant string, variable-length string, an array of character, or a character.

Return Value

`index` returns an integer value that represents the position of the first occurrence of *pattern_string* within *target_string*. If the first occurrence is at the starting position of the original string, the returned `index` value is 1.

Comments

The left-most occurrence of the *pattern-string* is considered the first occurrence.

If the *pattern_string* is not found in the *target_string*, `index` returns 0. If *pattern_string* is the null string, `index` returns –1.

Example

The Pascal program, `index.p`:

```
program index_example;

{ This program demonstrates the use of
  the index function. }

const
    MAX = 20;
    STRING = 'FOO';

type
    char_array = varying [MAX] of char;

var
    s1: char_array := 'INDEX_EXAMPLE';
    s2: char_array := 'EXAMPLE';
    i: integer16;

procedure print(index: integer; s1: char_array;
                s2: char_array);

begin
    if index = 0 then begin
      write('The string ', s2, ' is not');
      writeln(' in the string ', s1, '.')
    end else begin
      write('The string ', s2, ' is at index ', i: 1);
      writeln(' in the string ', s1, '.')
    end
end; { print } { print } { print }

begin { main program }
    i := index(s1, s2);
    print(i, s1, s2);
    i := index(s1, STRING);
    print(i, s1, STRING)
end. { index_example }
```

The commands to compile and execute `index.p`:

```
hostname% pc index.p
hostname% a.out
The string EXAMPLE is at index 7 in the string INDEX_EXAMPLE.
The string FOO is not in the string INDEX_EXAMPLE.
hostname%
```

land

The `land` function returns the bitwise `and` of two integer values.

Syntax

`land`(*int1*, *int2*)

Arguments

int1 and *int2* are integer expressions.

Return Value

`land` returns an integer value.

Comments

`land` performs a bit-by-bit `and` operation using Table 6-10.

Table 6-10 `land` Truth Table

Value of *bit in int1*	**Value of *bit in int2***	**Value of *bit in result***
0	0	0
0	1	0
1	0	0
1	1	1

If *int1* and *int2* are different size integers, Pascal converts the smaller integer to the larger integer before it performs the `land` operation.

land produces the same results as the bitwise operator &. Do not confuse land with the boolean operator and, which finds the logical and of two boolean expressions.

Example

The Pascal program, `land.p`:

```
program land_example;

{  This program demonstrates the use of the land, lor,
   lor, and xor functions. }

procedure BinaryOutput(intval: integer32);

var
    i: integer32;

begin
    write(' Decimal : ', intval, ' Binary : ');
    for i := 31 downto 0 do begin
      if lsr(intval, i) mod 2 = 0 then
          write('0')
      else
          write('1')
    end;
    writeln
end; { BinaryOutput }

var
    ival1, ival2: integer32;

begin
    ival1 := 2#00000000000000000000000000001111;
    ival2 := 2#00000000000000000000000011111111;
    writeln('IVAL1');
    BinaryOutput(ival1);
    writeln('IVAL2');
    BinaryOutput(ival2);
    writeln('LNOT(IVAL1)');
    BinaryOutput(lnot(ival1));
    writeln('LAND(IVAL1,IVAL2)');
    BinaryOutput(land(ival1, ival2));
    writeln('LOR(IVAL1,IVAL2)');
    BinaryOutput(lor(ival1, ival2));
    writeln('XOR(IVAL1,IVAL2)');
    BinaryOutput(xor(ival1, ival2))
end. { land_example }
```

The commands to compile and execute `land.p`. The value the bit-shift routines return may depend upon the architecture of your system.

```
hostname% pc land.p
hostname% a.out
IVAL1
 Decimal : 15 Binary : 00000000000000000000000000001111
IVAL2
 Decimal : 255 Binary :00000000000000000000000011111111
LNOT(IVAL1)
 Decimal : -16 Binary :11111111111111111111111111110000
LAND(IVAL1,IVAL2)
 Decimal : 15 Binary : 00000000000000000000000000001111
LOR(IVAL1,IVAL2)
 Decimal : 255 Binary :00000000000000000000000011111111
XOR(IVAL1,IVAL2)
 Decimal : 240 Binary :00000000000000000000000011110000
hostname%
```

lastof

The `lastof` function returns the value of the upper bound when its argument is or has an ordinal type. For array types, `lastof` returns the upper bound for the subrange defining the array index. For set types, it returns the upper bound of the set base type.

Syntax

`lastof`(*x*)

Arguments

x is either a variable, a constant, an expression, or the name of a user-defined or predeclared Pascal data type. *x* may not be a record, a file, a pointer type, a conformant array, a procedure or function parameter, or a string literal.

Return Value

When *x* is an ordinal type, a constant, an expression, or variable, the value `lastof` returns has the same data type as its argument.

When *x* is an array, the value `lastof` returns has the same data type as the type of the array index.

When *x* is a set type, the value `lastof` returns has the same data type as the base type of the set.

Comments

Pascal follows the rules in Table 6-11 when returning the value of *x*.

Table 6-11 `lastof` Return Values

Type of Argument	Return Value
`integer` (without –xl)	2147483647
`integer` (with –xl)	32767
`integer16`	32767
`integer32`	2147483647
`char`	chr(255)
`boolean`	true
`enumerated`	The last element in the enumeration type declaration.
`array`	The upper bound of the subrange that defines the array size.
`varying`	The upper bound of the `varying` string.
`set of 'A'..'Z'`	The character 'Z'.

Example

See the example in `Section , "firstof," on page 124`

length

The `length` function returns the length of a string.

Syntax

`length(`*str*`)`

Arguments

str is a variable-length string, a character array, or a character-string constant.

Return Value

`length` returns an integer value.

Comments

`length` returns a value that specifies the length of *str*.

Example

The Pascal program, `length.p`:

```
program length_example(output);

{ This program demonstrates the use of the length function. }

var
   s1: array [1..15] of char;
   s2: varying [20] of char;
begin
   s1 := 'San Francisco ';
   s2 := 'California';
   writeln('The length of string one is ', length(s1): 2, '.');
   writeln('The length of string two is ', length(s2): 2, '.');
   writeln('The combined length is ', length(s1 + s2): 2, '.')
end. { length_example }
```

The commands to compile and execute `length.p`:

```
hostname% pc length.p
hostname% a.out
The length of string one is 15.
The length of string two is 10.
The combined length is 25.
hostname%
```

linelimit

The `linelimit` procedure terminates execution of a program after a specified number of lines has been written into a text file.

Syntax

`linelimit(`*file*, *n*`)`

Arguments

file is a file having the `text` or `file` data type.

n is a positive integer expression.

Return Value

`linelimit` does not return any values.

Comments

`linelimit` terminates program execution if more than *n* lines are written to file *f*. If *n* is less than zero, no limit is imposed.

`linelimit` has no effect unless you compile your program with the, **`-C`** option.

Example

The Pascal program, `linelimit.p`:

```
program linelimit_example;

{ This program demonstrates the use of the
  linelimit procedure. }

const
    FILE = 'linelimit.dat';

var
    infile: text;
    error: integer32;
    name: array [1..20] of char;

begin
    open(infile, FILE, 'unknown', error);
    rewrite(infile, FILE);
    if error = 0 then begin
      writeln('Enter the names of your children.');
      writeln('The last entry should be "0".');
      repeat
          readln(name);
          writeln(infile, name);
          linelimit(infile, 10)
      until name = '0';
      close(infile)
    end else begin
      writeln('Difficulty opening file.');
      writeln('Error code = ', error, '.')
    end
end. { linelimit_example }
```

The commands to compile and execute `linelimit.p`:

```
hostname% pc -C linelimit.p
hostname% a.out
Enter the names of your children.
The last entry should be "0".
Ryan
Matthew
Jennifer
Lynne
Lisa
Ann
Katherine
Devon
Geoffrey
Brian

linelimit.dat : Line limit exceeded
*** a.out terminated by signal 5: SIGTRAP
*** Traceback being written to a.out.trace
Abort (core dumped)
hostname%
```

lnot

The `lnot` function returns the bitwise `not` of an integer value.

Syntax

`lnot(`*int*`)`

Arguments

int is an integer expression.

Return Value

`lnot` returns an integer value.

Comments

`lnot` performs a bit-by-bit `not` operation using Table 6-12.

Table 6-12 `lnot` Truth Table

Value of bit in int	Value of bit in result
0	1
1	0

`lnot` produces the same results as the bitwise operator ~. Do not confuse `lnot` with the boolean operator `not`, which evaluates the logical `not` of a boolean expression.

Example

See the example in Section , “land,” on page 139

lor

The `lor` function returns the inclusive `or` of two integer values.

Syntax

`lor(`*int1*, *int2*`)`

Argument

int1 and *int2* are integer expressions.

Return Value

`lor` returns an integer value.

Comments

`lor` performs an inclusive `or` using Table 6-13.

Table 6-13 `lor` Truth Table

Value of bin in intl1	Value of bin in int2	Value of bit in result
0	0	0
0	1	1
1	0	1
1	1	1

If *int1* and *int2* are different size integers, Pascal converts the smaller integer to the larger integer before it performs the `lor` operation.

`lor` produces the same results as the bitwise operators `!` and `|`. Do not confuse `lor` with the boolean operator `or`, which evaluates the logical `or` of a boolean expression.

Example

See the example in the `land` listing earlier in this chapter.

lshft

The `lshft` function does a logical left shift of an integer value.

Syntax

`lshft(`*num*,*sh*`)`

Argument

num and *sh* are integer expressions.

Return Value

`lshft` returns a 32-bit integer value.

Comments

`lshft` shifts all bits in *num sh* places to the left. `lshft` does not wrap bits from the left to right. The value `lshft` returns is machine-dependent and is unspecified unless 0 <= *sh* <= 32.

Do not confuse `lshft` with the arithmetic left shift functions which preserve the sign bit.

Example
The Pascal program, `lshft.p`:

```
program lshft_example(input, output);

{ This program does a logical left shift. }

const
    SIZE = 8;

var
    i: integer32;
    i32: integer32;
    loop: integer32;

begin
    write('Enter a positive or negative integer:  ');
    readln(i);
    for loop := 1 to SIZE do begin
      i32 := lshft(i, loop);
      write('Logical left shift ', loop: 2);
      writeln(' bit(s):  ', i32 hex)
    end
end. { lshft_example }
```

The commands to compile and execute `lshft.p`. The value the bit-shift routines return may depend upon the architecture of your system.

```
hostname% pc lshft.p
hostname% a.out
Enter a positive or negative integer: 3
Logical left shift 1 bit(s):   6
Logical left shift 2 bit(s):   C
Logical left shift 3 bit(s):  18
Logical left shift 4 bit(s):  30
Logical left shift 5 bit(s):  60
Logical left shift 6 bit(s):  C0
Logical left shift 7 bit(s): 180
Logical left shift 8 bit(s): 300
hostname%
```

lsl

The `lsl` function is identical to the `lshft` function. See the `lshft` listing earlier in this chapter.

lsr

The `lsr` function is identical to the `rshft` function. See the `rshft` listing later in this chapter.

max

The `max` function evaluates two scalar expressions and returns the larger one.

Syntax

`max(`*exp1*, *exp2*`)`

Arguments

exp1 and *exp2* are any valid scalar expressions that are assignment compatible.

Return Value

`max` returns the same or the converted type of *exp1* and *exp2*.

See also the `min` listing.

Example

The Pascal program, `max.p`:

```
program max_example(input, output);

{ This program reads in 10 positive integers
  in the range 1 through 501 and determines
  the largest even and smallest odd.Out of range numbers
  are rejected. }

var
   smallest_odd: integer := 501;
   largest_even: integer := 0;
   number, counter: integer;

begin
 writeln('Please enter 10 integers between 0 and 501:');
 for counter := 1 to 10 do begin
  read(number);
  if (number < 0) or (number > 501)
    then writeln ('The number is out of range ')
    else if odd(number)
      then smallest_odd := min(number, smallest_odd)
      else
          largest_even := max(number, largest_even)
   end;
   writeln('The smallest odd number is ', smallest_odd: 1, '.');
   writeln('The largest even number is ', largest_even: 1, '.')
end. { max_example }
```

The commands to compile and execute `max.p`:

```
hostname% pc max.p
hostname% a.out
Please enter 10 integers between 0 and 501:
56 431 23 88 222 67 131 337 401 99
The smallest odd number is 23.
The largest even number is 222.
hostname%
```

message

The `message` procedure writes the specified information on `stderr` (usually the terminal).

Syntax

`message(`*x1*, ..., *xN*)

Arguments

x is one or more expressions separated by commas. *x* may be a variable, constant, or expression of a type that `write` accepts (such as integer, real, character, boolean, enumerated, or string). *x* cannot be a set variable.

Return Value

`message` does not return any values.

Comments

`message` is an output procedure similar to `write` and `writeln`. Whereas `write` and `writeln` send the output to standard output or the specified file, `message` sends the output to standard error. `message` also appends a carriage return to the message.

`message` flushes all buffers both before and after writing the message.

`message(`*x1*, ..., *xN*) is equivalent to the following code:

```
writeln(errout, x1, ..., xN);
flush(errout);
flush(output);
    .
    .
 { Flush all open files. }
```

Example

The Pascal program, `message.p`:

```
program message_example(output);

{ This program demonstrates the use of the
  message function. }

begin
    writeln('This message will go to standard output.');
    message('This message will go to standard error.')
end. { message_example }
```

The commands to compile and execute `message.p`:

```
hostname% pc message.p
hostname% a.out > temp_file
This message will go to standard error.
hostname% cat temp_file
This message will go to standard output.
hostname% a.out >& temp_file
hostname% cat temp_file
This message will go to standard output.
This message will go to standard error.
hostname%
```

min

The `min` function evaluates two scalar expressions and returns the smaller one.

Syntax

`min(`*exp1*, *exp2*`)`

Arguments

exp1 and *exp2* are any valid scalar expressions that are assignment compatible.

Return Value

`min` returns the same or the converted type of *exp1* and *exp2*.

Comments

See also the `max` listing.

Example

See the example under the `max` listing earlier in this chapter.

null

The `null` procedure performs no operation.

Syntax

`null`

Arguments

`null` does not take any arguments.

Return Value

`null` does not return any values.

Comments

`null` does absolutely nothing; it is useful as a placeholder. For example, suppose you are developing a program, and you are uncertain about a particular `case` statement. You could put `null` in place of the `case` statement and later replace it with an actual function or procedure.

open

The `open` procedure associates an external file with a file variable.

Syntax

`open(`*file, pathname, history, error, buffer*`)`

Arguments

file is a variable having the `text` or `file` data type.

pathname is a string constant or string variable.

history is a string variable.

error is an `integer32` variable. This argument is optional.

buffer is an optional integer variable. This argument is currently ignored.

Return Value

`open` does not return any values.

Comments

`open` associates the permanent file *file* with a file variable for reading or writing. `open` does *not* actually open the file; you must call `reset` or `rewrite` before reading or writing to that file.

pathname must be one of the following:

- An operating system pathname.
- A string of `'^`*n*`'`, where *n* is an integer from 1 to 9. *n* represents the *n*th argument passed to the program. ^n is equivalent to `argv(n, file)`.
- A prompt string. The string must begin with the character '*'. Pascal prints the prompt string on the standard output at runtime.
- The string `'-STDIN'` or `'-STDOUT.'`[1]
- A variable or constant that contains any of the above items.

history tells the compiler whether to create the file or what to do with it if it exists. *history* must be one of these values:

1. This feature is not implemented in the current version of the product

`'new'` tells the compiler to associate the operating system file with a new file. The compiler generates an error if the file already exists.

`'old'` tells the compiler to associate the operating system file with an existing file. The compiler generates an error if the file does not exist.[1]

`'unknown'` tells the compiler to search for an existing file and associate it with the operating system file. The compiler creates the file if it does not exist.[2]

Pascal returns an integer error code through *error* as shown in Table 6-14.

Table 6-14 `open` Error Codes

Number	Description
0	open was successful
1	File not specified on the command line. For example, this error is generated for the following line when argument one is not specified: `open(infile,'^1','new',Error);`
2	Unable to open file.
3	Invalid *history* specified. *history* must be either '`new`', '`old`', or '`unknown`'.

Pascal automatically closes all open files when your program terminates or when the program exits the scope in which the file variable for the open file is allocated. See also the `close`, `reset`, and `rewrite` procedures.

1. This option will first try to open the file for writing. If it can't, it will try to open it for reading only.

The Pascal program, `open.p`:

```
program open_example;

{ This program demonstrates the use of the open procedure. }

const
    name_of_file = 'open1.txt';
    file3 = '*Enter_a_filename-- ';

type
    char_array = varying [50] of char;

var
    infile: text;
    error: integer32;
    name: char_array;

begin
    { Open an existing file. }
    open(infile, name_of_file, 'old', error);
    if error = 0 then begin
      writeln('Opened ', name_of_file, ' for reading.');
      close(infile)
    end else
      writeln('Error opening file', name_of_file, error);

    { Open a file specified by a command line argument. }
    open(infile, '^1', 'unknown', error);
    if error = 0 then begin
      argv(1, name);
      writeln('Opened ', name, ' for reading.');
      close(infile)
    end else
      writeln('No command line argument; error code =', error);

    { Open a file that may or may not exist. }
    { Prompt user for name of file at runtime. }
    open(infile, file3, 'unknown', error);
    if error = 0 then begin
      writeln('Opened file for reading.');
      close(infile)
    end else
      writeln('Error opening file', error)
end. { open_example }
```

The commands to compile and execute `open.p`:

```
hostname% pc open.p
hostname% a.out
Opened open1.txt for reading.
No command line argument; error code =     1
Enter_a_filename-- test.txt
Opened file for reading.
hostname%
```

pcexit

The `pcexit` function does the following:

1. Checks whether any imposed statement count has been exceeded.
2. Calls the `ieee_retrospective ( )` routine. See the SunOS 5.0 operating system document the *Numerical Computation Guide.*
3. Terminate the program with the specified return value (similar to the C `exit()` function).

Syntax

`pcexit`(*x*)

Arguments

x is an integer variable or constant.

Return Value

`pcexit` does not return any values.

Comments

The C function `exit` (3C) calls any functions registered through the `atexit`(3C) function in the in the reversed order of their registration.

6

random

The `random` function generates a random number between 0.0 and 1.0.

Syntax

`random`(*x*)

Arguments

x has no significance and is ignored.

Return Value

`random` returns a real value.

Comments

`random` generates the same sequence of numbers each time you run the program. See the `seed` function to reseed the number generator.

Example

See the examples in the following page.

The Pascal program, `random.p`:

```
program random_example(output);

{ This program demonstrates the use of
  the random function. }

var
    i: integer;
    x: integer;

begin
    writeln('These numbers were generated at random:');
    for i := 1 to 5 do begin
      write(trunc(random(x) * 101))
    end;
    writeln
end. { random_example }
```

The commands to compile and execute `random.p`:

```
hostname% pc random.p
hostname% a.out
These numbers were generated at random:
    97    6    48    91    35
hostname%
```

read and readln

Pascal supports the standard form of `read` and `readln` with three extensions:

- Read in boolean variables.
- Read in fixed- and variable-length strings.
- Read in enumerated type values from a text file.

Syntax

`read`(*file, var1 ..., varN*);

`readln`(*file, var1 ..., varN*);

Arguments

file is an optional variable having either the `text` or `file` data type.

var can be any real, integer, character, boolean, subrange, enumerated, or array variable or a fixed- or variable-length string variable. If `read` or `readln` are used in a function to define the function result, var can also be an identifier of that function.

Return Value

`read` and `readln` do not return any values.

Comments

If *var* is a variable-length string, `read` and `readln` try to read in as many characters as indicated by the current length, up to the first newline character. `read` and `readln` do not pad the string with blanks if the length of the string is less than the current length.

With both variable- and fixed-length strings, if the number of characters on a line is more than the maximum length of the string, the next `read` will pick up where the last `read` left off. With `readln`, the rest of the line is thrown out, so the next `read` or `readln` begins at the next line.

If *var* is an enumerated type, `read` and `readln` attempt to read a value that is included in the type definition. If the value is not in the type definition, the compiler terminates program execution and prints the following message:

```
Unknown name "value" found on enumerated type read
Trace/BPT trap (core dumped)
```

You may trap this error with the I/O error recovery mechanism, described in Section , “I/O Error Recovery,” on page 204. Using `read` or `readln` in the block of a function in the form:

```
read (..., f, ...)
```

is treated as if it were an assignment of the form:

```
f:=e
```

where e is the input value. This feature is an extension of the Pascal Standard and so cannot be used with the -s option.

Example

See the examples in the following page.

The Pascal program, `read.p`

```
program read_example(input, output);

{ This program uses readln to input strings,
  boolean data, and enumerated data. }

type
    gem_cuts = (marquis, emerald, round, pear_shaped);

var
    x: gem_cuts;
    gem: varying [10] of char;
    gift: boolean;

begin
    write('Enter type of gem:  ');
    readln(gem);
    write('Enter cut: ');
    write('marquis, emerald, round, pear_shaped:  ');
    readln(x);
     write('Enter true if this a gift, false if it is not:  ');
    readln(gift);
    write('You have selected a ', gem);
    writeln(' with a ', x, ' cut.');
    if gift then
      writeln('We will gift wrap your purchase for you.')
end. { read_example }
```

The commands to compile and execute `read.p`:

```
hostname% pc read.p
hostname% a.out
Enter type of gem: diamond
Enter cut: marquis, emerald, round, pear_shaped: pear_shaped
Enter true if this a gift, false if it is not: true
You have selected a diamond with a pear_shaped cut.
We will gift wrap your purchase for you.
hostname%
```

remove

The `remove` procedure removes the specified file.

Syntax

`remove(`*file*`)`

Arguments

file is either a fixed- or variable-length string that indicates the name of the file to be removed. *file* cannot be a `text` or `file` variable.

Return Value

`remove` does not return any values.

Comments

Pascal generates an I/O error if the file does not exist. You may trap this error with the I/O error recovery mechanism, described in Section , “I/O Error Recovery,” on page 204

Example

See the examples in the following page.

The Pascal program, `remove.p`:

```
program remove_example;

{ This program demonstrates the use of the
  remove procedure. }

var
    name: varying [10] of char;

begin
    if argc <> 2 then
      writeln('Usage is : rm <file>')
    else begin
      argv(1, name);
      remove(name)
    end
end. { remove_example }
```

The commands to compile and execute `remove.p`:

```
hostname% pc remove.p
hostname% touch rmc
hostname% ls rmc
rmc
hostname% a.out rmc
hostname% ls rmc
rmc not found
```

reset

Pascal supports an optional second argument to the `reset` procedure. This argument gives an operating system file name.

Syntax

`reset(`*file, filename*`)`

Arguments

file is a variable having the `text` or `file` data type.

filename is a fixed- or variable-length string, or a string constant.

Return Value

`reset` does not return any values.

Comments

`reset` tells the system to allow you to read from the file. `reset` does not allow you to write to the file.

In standard Pascal, `reset` takes only one argument, a file variable. In Pascal, `reset` can take an optional second argument, an operating system file name. If you give the optional file name, the compiler opens the file with that name on the current path and associates it with the given file variable.

For example, this code associates the Pascal file `data` with the operating system file `primes`:

```
reset(data, 'primes');
```

`reset` does an implicit close on the file, thereby allowing you to reuse its file variable with a different file. Similarly, if `input` or `output` are reset the current implementation of the product also implicitly closes `stdin` and `stdout`.

`reset` normally generates an error and halts if the file specified in the two argument form does not exist. You may trap this error with the I/O error recovery mechanism, described in Section , "I/O Error Recovery," on page 204

See also `rewrite`, which opens a file for writing.

Example

See the example in the `rewrite` listing.

rewrite

Pascal supports an optional second argument to the `rewrite` procedure. This argument gives an operating system file name.

Syntax

`rewrite(`*file*, *filename*`)`

Arguments

file is a variable having the `text` or `file` data type.

filename is a fixed- or variable-length string, or a string constant.

Return Value

`rewrite` does not return any values.

Comments

`rewrite` tells the system to allow you to modify a file.

- In standard Pascal, `rewrite` takes only one argument, a file variable.
- In Pascal, `rewrite` can take an optional second argument, an operating system file name.

In Pascal, if you give the optional file name, the compiler opens the file with that name on the current path and associates it with the given file variable. For example, this code associates the Pascal file `data` with the operating system file `primes`:

```
rewrite(data, 'primes');
```

If you do not give an optional second argument, Pascal creates a physical operating system file for you. This file has the same name as the file variable *if* the file variable is listed in the program header. If the file variable is *not* listed in the program header, Pascal creates a temporary file with the name `#tmp.`*suffix*. The temporary file is deleted when the program terminates.

If the file variable is `output`, and the second argument is not given, Pascal creates a temporary file, but does not delete it after the program exits.

`rewrite` does an implicit close on the file, thereby allowing you to reuse its file variable with a different file.

See also `reset`, which opens a file for reading.

Example

The Pascal program, `rewrite.p`:

```
program rewrite_example(input, output);

{ This program demonstrates the use of rewrite
  and reset.}

const
    MAX = 80;

var
    f: text;
    line: varying [MAX] of char;

begin
    rewrite(f, 'poem.txt');
    write('Enter a line of text.');
    writeln('  Hit Control-D to end the job.');
    while not eof do begin
      readln(line);
      writeln(f, line)
    end;
    close(f);
    writeln;
    writeln;
    writeln('These are the lines of text you input:');
    reset(f, 'poem.txt');
    while not eof(f) do begin
      readln(f, line);
      writeln(line)
    end;
    close(f)
end. { rewrite_example }
```

The commands to compile and execute `rewrite.p`:

```
hostname% pc rewrite.p
hostname%  a.out
Enter a line of text. Hit Control-D to end the job.
Hello, how are you?
Please keep in touch.
 ^D

These are the lines of text you input:
Hello, how are you?
Please keep in touch.
hostname%
```

rshft

The `rshft` function does a logical right shift of an integer value.

Syntax

`rshft(`*num*, *sh*`)`

Arguments

num and *sh* are integer expressions.

Return Value

`rshft` returns a 32-bit integer value.

Comments

`rshft` shifts the bits in *num sh* spaces to the right. `rshft` does not preserve the sign bit (left-most) bit of a number and does not wrap bits from right to left. The value `rshft` returns is machine-dependent and is unspecified unless 0 <= *sh* <= 32. Do not confuse `rshft` with the arithmetic right shift functions `asr` and `arshft`, which preserve the sign bit.

The Pascal program, `rshft.p`:

```
program rshft_example(input, output);

{ This program demonstrates the logical right shift. }

const
    SIZE = 8;

var
    i: integer32;
    i32: integer32;
    loop: integer32;

begin
    write('Enter a positive or negative integer:  ');
    readln(i);
    for loop := 1 to SIZE do begin
      i32 := rshft(i, loop);
      write('Logical right shift ', loop: 2);
      writeln(' bit(s):  ', i32 hex)
    end
end. { rshft_example }
```

The commands to compile and execute `rshft.p`. The value the bit-shift routines return may depend upon the architecture of your system.

```
hostname% pc rshft.p
hostname% a.out
Enter a positive or negative integer: 32
Logical right shift 1 bit(s):        10
Logical right shift 2 bit(s):         8
Logical right shift 3 bit(s):         4
Logical right shift 4 bit(s):         2
Logical right shift 5 bit(s):         1
Logical right shift 6 bit(s):         0
Logical right shift 7 bit(s):         0
Logical right shift 8 bit(s):         0
hostname%
```

seed

The `seed` function reseeds the random number generator.

Syntax

`seed(`*x*`)`

Arguments

x is an integer value.

Return Value

`seed` returns an integer value.

Comments

`seed` sets the random number generator to *x* and returns the previous seed. If you do not reseed the random number generator, the `random` function returns the same sequence of random numbers each time you run the program. To produce a different random number (sequence each time the program is run), set the seed with the statement:

```
x := seed(wallclock);
```

Example

The Pascal program, `seed.p`:

```
program seed_example(output);

{ This program demonstrates the use of the
  seed function. }

var
    i: integer;
    x: integer;
begin
    x := seed(wallclock);
    writeln('These numbers were generated at random:');
    for i := 1 to 5 do begin
      write(trunc(random(i) * (i * 101)))
    end;
    writeln
end. { seed_example }
```

The commands to compile and execute `seed.p`:

```
hostname% pc seed.p
hostname% a.out
These numbers were generated at random:
   75   175   186   260   178
hostname%
```

sizeof

`sizeof` returns the number of bytes the program uses to store a data object.

Syntax

`sizeof(`*x*, *tag1*, ... *tagN*`)`

Arguments

x is any predeclared or user-defined Pascal data type, a variable, a constant, or a string.

tag is a constant. *tag* is optional.

Return Value

`sizeof` returns an integer value.

Comments

`sizeof` returns the number of bytes in the data object *x*. The *tag*s correspond to the fields in a variant record. The *tag*s are effectively ignored because Pascal allocates records according to the largest variant.

You cannot use `sizeof` to determine the size of a conformant array parameter because the array size is not known until run time.

Example

The Pascal program, `sizeof.p`:

```
program sizeof_example(output);

{ This program demonstrates the use of the
  sizeof function. }

const
    MAX = 5;

type
    subB = false..true;
    sub1 = 0..7;
    sub2 = 0..127;
    sub3 = 0..255;
    color1 = (re, gree, blu, whit);
    color2 = (red, green, blue, white, orange, purple, black);
    rec_type =
      record
          i: integer;
          ar: array [1..MAX] of single;
          d: double
      end;

var
    b: boolean;
    c: char;
    f: text;
    i: integer;
    i16: integer16;
    i32: integer32;
    s: shortreal;
    r: real;
    l: longreal;
    rec: rec_type;
    u: univ_ptr;
```

`sizeof.p` program (Screen 1 of 2)

```
begin
    writeln('The size of boolean is    ', sizeof(b), '.');
    writeln('The size of char is       ', sizeof(c), '.');
    writeln('The size of color1 is     ', sizeof(color1), '.');
    writeln('The size of color2 is     ', sizeof(color2), '.');
    writeln('The size of file is       ', sizeof(f), '.');
    writeln('The size of integer is    ', sizeof(i), '.');
    writeln('The size of integer16 is  ', sizeof(i16), '.');
    writeln('The size of integer32 is  ', sizeof(i32), '.');
    writeln('The size of longreal is   ', sizeof(l), '.');
    writeln('The size of shortreal is  ', sizeof(s), '.');
    writeln('The size of real is       ', sizeof(r), '.');
     writeln('The size of rec_type is   ', sizeof(rec_type), '.');
    writeln('The size of rec_type.ar   ', sizeof(rec.ar), '.');
    writeln('The size of subB          ', sizeof(subB), '.');
    writeln('The size of sub1 (8)      ', sizeof(sub1), '.');
    writeln('The size of sub2 (128)    ', sizeof(sub2), '.');
    writeln('The size of sub3 (256)    ', sizeof(sub3), '.');
    writeln('The size of univ_ptr      ', sizeof(u), '.')
end. { sizeof_example }
```

`sizeof.p` program (Screen 2 of 2)

The commands to compile and execute `sizeof.p`. The value `sizeof` returns may depend upon the architecture of your system.

```
hostname% pc sizeof.p
hostname% a.out
The size of boolean is1.
The size of char is1.
The size of color1 is1.
The size of color2 is1.
The size of file is2089.
The size of integer is 4.
The size of integer16 is 2.
The size of integer32 is 4.
The size of longreal is 8.
The size of shortreal is 4.
The size of real is 8.
The size of rec_type is 32.
The size of rec_type.ar 20.
The size of subB 1.
The size of sub1 (8) 1.
The size of sub2 (128) 1.
The size of sub3 (256) 2.
The size of univ_ptr 4.
hostname%
```

stlimit

The `stlimit` procedure terminates program execution if a specified number of statements have been executed in the current loop.

Syntax

`stlimit(`*x*`)`

Arguments

x is an integer value.

Return Value

`stlimit` does not return any values.

Comments

To use `stlimit`, you must include the following in your source program:

```
{$p+}
```

When you call `stlimit`, it tests if *x* number of statements have been executed in the current loop. If the number of statements executed equals or exceeds *x*, `stlimit` stops the program, dumps core, and prints the following message:

```
Statement count limit of x exceeded
Trace/BPT trap (core dumped)
```

If `stlimit` is used without a loop, it reports the number of statements executed and the CPU time utilized.

If you want to check the statement limit after each statement, you can turn on runtime checks using the `-C` command line option or the `C` or `t` program text options. When runtime checks are turned on and the compiler encounters a `stlimit` statement, the compiler inserts a statement limit check after each subsequent statement.

Example

The Pascal program, `stlimit.p`:

```
program stlimit_example;
{$p+}

{ This program demonstrates the use
  of the stlimit procedure. }

begin
    repeat
      writeln('Hello.');
      stlimit(10)
    until false
end. { stlimit_example }
```

The commands to compile and execute `stlimit.p`:

```
hostname% pc stlimit.p
hostname% a.out
Hello.
Hello.
Hello.
Hello.

Statement count limit of 11 exceeded
Trace/BPT trap (core dumped)
hostname%
```

substr

The `substr` function takes a substring from a string.

Syntax

`substr(`*str, p, n*)

Arguments

str is a variable-length string, a character array, or a character-string constant.

p is a positive integer.

n is a positive integer. *n* must be a positive integer.

Return Value

`substr` returns a variable-length string.

Comments

`substr` returns a substring beginning at position `p` and continuing for `n` characters. If the values of either `p` or `n` indicate a character outside the bounds of the string size, Pascal returns a null string.

Example

The Pascal program, `substr.p`:

```
program substr_example(output);

{ This program demonstrates the use of the
  substr function. }

var
    string1: array [1..15] of char;
    string2: varying [25] of char;

begin
    string1 := 'Paris, Texas';
    string2 := 'Versailles, France';
    write(substr(string1, 1, 6));
    writeln(substr(string2, 12, 7))
end. { substr_example }
```

The commands to compile and execute `substr.p`:

```
hostname% pc substr.p
hostname% a.out
Paris, France
hostname%
```

sysclock

The `sysclock` function returns the system time consumed by the process.

Syntax

```
sysclock
```

Arguments

`sysclock` does not take any arguments.

Return Value

`sysclock` returns an integer value.

Comments

`sysclock` returns the system time in milliseconds. See also the `clock` function, which returns the user time the process consumes.

Example

See the example in the `clock` listing earlier in this chapter.

time

The `time` procedure retrieves the current time.

Syntax

`time(`*a*`)`

Arguments

a is either a character array 10 elements long or a variable-length string.

Return Value

`time` returns a character string in the form `hh:mm:ss`, where `hh` is the hour (0 through 23), `mm` is minutes (0 through 59), and `ss` is seconds (0 through 59).

Comments

`time` uses a 24-hour clock; 8:00 p.m., for example, is shown as 20:00:00. `time` puts a zero in front of the hours, minutes, and seconds to make them have two digits. `time` also puts a blank at the beginning and end of the string.

Example

The Pascal program, `time.p`:

```
program time_example(output);

{ This program retrieves the current time. }

var
    s1: alfa;
    s2: array [1..10] of char;
    s3: array [89..98] of char;
    s4: varying [100] of char;

begin
    time(s1);
    time(s2);
    time(s3);
    time(s4);
    writeln('The time is ', s1, '.');
    writeln('The time is ', s2, '.');
    writeln('The time is ', s3, '.');
    writeln('The time is ', s4, '.');
end. { time_example }
```

The commands to compile and execute `time.p`:

```
hostname% pc time.p
hostname% a.out
The time is 15:40:17 .
The time is 15:40:17 .
The time is 15:40:17 .
The time is 15:40:17 .
hostname%
```

trim

The `trim` function removes the trailing blanks in a character string.

Syntax

`trim`(*input_string*)

Arguments

input_string is a constant string, a variable-length string, a character array, or a character.

Return Value

`trim` returns a variable-length string equal to the input string without any trailing blanks. If *input_string* is a `null` string or contains only blanks, `trim` returns a `null` string of length 0.

Comments

`trim` has no effect if its result value is assigned to a fixed-length character string variable. Fixed-length characters are always padded with blanks during assignments.

Example

See the example in the following page.

The Pascal program, `trim.p`:

```
program trim_example;

{ This program demonstrates the use of the trim function. }
const
    TEN = '          ';
    MAX = 10;
type
    large = varying [100] of char;
    s_type = array [1..MAX] of char;
    v_type = varying [MAX] of char;
var
    c1: char := ' ';
    st1: s_type := '123456    ';
    st2: s_type := '          ';
    st3: s_type := '0123456789';
    v1: v_type := '01234     ';
    v2: v_type := '          ';
    v3: v_type := '0123456789';
    l: large;

begin
    l := trim(st1) + trim(st2) + trim(st3) + trim(c1);
    writeln(l, length(l));
    l := substr(trim(st1) + trim(st2) + trim(st3), 3, 5);
    writeln(l, length(l));
    l := trim(v1) + trim(TEN) + trim(v2) + trim(v3) + trim(st1) + trim(st2) + trim(st3);
    writeln(l, length(l))
end. { trim_example }
```

The commands to compile and execute `trim.p`:

```
hostname% pc trim.p
hostname% a.out
1234560123456789         16
34560         5
01234012345678912345601234567891         31
hostname%
```

type transfer

The type transfer functions change the data type of a variable, constant, or expression.

Syntax

transfer_function(*x*)

Arguments

transfer_function is a predeclared or user-defined Pascal data type.

x is a variable, constant, or expression.

Return Value

A type transfer function returns its argument unchanged in internal value, but with a different apparent type.

Comments

Suppose your program contains the following data type declarations:

```
var
  x: integer32;
  y: single;
```

To transfer the value of variable `x` to a floating-point number, you would write:

```
y := single(x);
```

When the argument of a type transfer function is a variable, the size of the argument must be the same as the size of the destination type. However, if the argument to a transfer function is a constant or an expression, Pascal attempts to convert the argument to the destination type because constants and expressions do not have explicit types.

The type transfer functions copy, but do *not* convert, any value. Do not confuse the type transfer functions with functions that actually convert the value of the variable, such as `ord`, `chr`, and `trunc`.

Example

The Pascal program, `type.p`:

```
program type_transfer_example(output);

{ This program uses transfer functions to
  convert integer to character. }

type
    range = 65..90;

var
    i: range;
    c: char;

begin
    for i := firstof(i) to lastof(i) do begin
      write('The character value of ', i: 1);
      writeln(' is ', char(i), '.')
    end
end. { type_transfer_example }
```

The commands to compile and execute `type.p`:

```
hostname% pc type.p
hostname% a.out
The character value of 65 is A.
The character value of 66 is B.
The character value of 67 is C.
The character value of 68 is D.
The character value of 69 is E.
The character value of 70 is F.
The character value of 71 is G.
The character value of 72 is H.
The character value of 73 is I.
The character value of 74 is J.
The character value of 75 is K.
The character value of 76 is L.
The character value of 77 is M.
The character value of 78 is N.
The character value of 79 is O.
The character value of 80 is P.
The character value of 81 is Q.
The character value of 82 is R.
The character value of 83 is S.
The character value of 84 is T.
The character value of 85 is U.
The character value of 86 is V.
The character value of 87 is W.
The character value of 88 is X.
The character value of 89 is Y.
The character value of 90 is Z.
hostname%
```

wallclock

The `wallclock` function returns the elapsed number of seconds since 00:00:00 GMT January 1, 1970.

Syntax

```
wallclock
```

Arguments

`wallclock` does not take any arguments.

Return Value

`wallclock` returns an integer value.

Comments

`wallclock` may be used with the `seed` function to generate a random number. It may also be used to time programs or parts of programs.

Example

The Pascal program, `wallclock.p`:

```
program wallclock_example(output);

{ This program demonstrates the use of the
  wallclock function. }

const
    NTIMES = 20;  { Number of times to compute Fib value. }
    NUMBER = 24;  { Biggest one we can compute with 16 bits. }

var
    start: integer;
    finish: integer;
    i: integer;
    value: integer;

{ Compute fibonacci number recursively. }
function fib(number: integer): integer;

begin
    if number > 2 then
      fib := fib(number - 1) + fib(number - 2)
    else
      fib := 1
end; { fib }

begin { main program }
    writeln('Begin computing fibonacci series.');
    write(NTIMES, ' Iterations:  ');
    start := wallclock;
    for i := 1 to NTIMES do
      value := fib(NUMBER);
    finish := wallclock;
    writeln('Fibonacci(', NUMBER: 2, ') = ', value: 4, '.');
    writeln('Elapsed time is ', finish - start: 3, ' seconds.')
end. { wallclock_example }
```

The commands to compile and execute `wallclock.p`:

```
hostname% pc wallclock.p
hostname% a.out
Begin computing fibonacci series.
    20 Iterations: Fibonacci(24) = 46368.
Elapsed time is   5 seconds.
hostname%
```

write and writeln

Pascal supports the standard form of `write` and `writeln` with the following extensions:

- Output enumerated type values to a text file.
- Write the internal representation of an expression in octal or hexadecimal.
- Specify a negative field width.

Syntax

`write`(*file, exp1:width ..., expN:width*)

`writeln`(*file, exp1:width ..., expN:width*)

Arguments

file is a variable having either the `text` or `file` data type. *file* is optional; it defaults to `output`.

exp is a variable, constant, or expression of type integer, real, character, boolean, enumerated, or string. *exp* cannot be a set variable.

width is an integer. *width* is optional.

Comments

If *exp* is an enumerated type, `write` and `writeln` attempt to write a value that is included in the type definition. If the value is not in the type definition, the compiler terminates program execution and prints an error message.

To write the internal representation of an expression in octal, use this form:

`write(`*x* `oct);`

x is a boolean, character, integer, pointer, or user-defined type. `x` can also be a constant, expression, or variable.

To write an expression in hexadecimal, use this form:

`write(`*x* `hex);`

When you specify a negative field width of a parameter, Pascal truncates all trailing blanks in the array. `write` and `writeln` assume the default values in Table 6-15 if you do not specify a minimum field length.

Table 6-15 Default Field Widths

Data Type	**Default Width without `-xl` option**	**Default Width with `-xl` option**
`array of char`	Declared length of the array	Declared length of the array
`boolean`	length of true or false	15
`char`	1	1
`double`	21	21
enumerated	length of type	15
hexadecimal	10	10
`integer`	10	10
`integer16`	10	10
`integer32`	10	10
`longreal`	21	21
`octal`	10	10
`real`	21	13
`shortreal`	13	13
`single`	13	13
String constant	No. of characters in string	No. of characters in string
Variable-length string	Current length of the string	Current length of the string

Example

The Pascal program, `octal.p`:

```
program octal_example(output);

{ This program writes a number in octal
  and hexadecimal format. }

var
    x: integer16;

begin
    write('Enter an integer:  ');
    readln(x);
    writeln(x: 5, ' is ', x oct, ' in octal.');
    writeln(x: 5, ' is ', x hex, ' in hexadecimal.')
end. { octal_example }
```

The commands to compile and execute `octal.p`:

```
hostname% pc octal.p
hostname% a.out
Enter an integer: 10
 10 is       12 in octal.
 10 is       A in hexadecimal.
hostname%
```

xor

The `xor` function returns the exclusive `or` of two integer values.

Syntax

`xor(`*int1*, *int2*`)`

Arguments

int1 and *int2* are integer expressions.

Return Value

`xor` returns an integer value.

Comments

Pascal uses Table 6-16 to return the bitwise exclusive `or` of *int1* and *int2*.

Table 6-16 `xor` Truth Table

Value of bin in intl1	**Value of bin in *int2***	**Value of bit in result**
0	0	0
0	1	1
1	0	1
1	1	1

If *int1* and *int2* are different size integers, Pascal converts the smaller integer to the larger integer before the it performs the `xor` operation.

`xor` is a bitwise operator similar to `&`, `!`, and `~`. Do not confuse it with the boolean operators `and`, `or`, and `not`.

Example

See the example in the `land` listing earlier in this chapter.

Input and Output 7

This chapter describes the Pascal input/output environment, with special attention to interactive programming.

Input/Output Routines

Pascal supports all standard input/output routines plus the extensions listed in Table 7-1. For a complete description of the routines, refer to Chapter 6, “Built-in Procedures and Functions.”

Table 7-1 Extensions to Input/Output Routines

Routine	Description
`close`	Closes a file.
`flush`	Writes the output buffer for the specified Pascal file into the associated operating system file.
`getfile`	Returns a pointer to the C standard I/O descriptor associated with the specified Pascal file.
`linelimit`	Terminates program execution after a specified number of lines has been written into a text file.
`message`	Writes the specified information to `stderr`.
`open`	Associates an external file with a file variable.
`read` and `readln`	Read in boolean, integer and floating-point variables, fixed- and variable-length strings, enumerated types, and pointers.
`remove`	Removes the specified file.
`reset` and `rewrite`	Accept an optional second argument.
`write` and `writeln`	Output boolean integer and floating-point variables, fixed- and variable-length strings, enumerated types, and pointers; output expressions in octal or hexadecimal; allow negative field widths.

The `eof` *and* `eoln` *Functions*

An extremely common problem encountered by new users of Pascal, especially in the interactive environment of the operating system, relates to `eof` and `eoln`. These functions are supposed to be defined at the beginning of execution of a Pascal program, indicating whether the input device is at the end of a line (`eoln`) or the end of a file (`eof`).

Setting `eof` or `eoln` actually corresponds to an implicit read in which the input is inspected, but not “used up”. In fact, there is no way the system can know whether the input is at the end of a file or the end of a line unless it attempts to read a line from it.

If the input is from a previously created file, then this reading can take place without run time action by the user. However, if the input is from a terminal, then the input is what you type. If the system does an initial read automatically at the beginning of program execution, and if the input is a terminal, you would have to type some input before execution could begin. This would make it impossible for the program to begin by prompting for input.

Pascal has been designed so that an initial read is not necessary. At any given time, Pascal may or may not know whether the end of file and end of line conditions are `true`.

Thus, internally, these functions can have three values: `true`, `false`, and "*I don't know yet; if you ask me I'll have to find out*." All files remain in this last, indeterminate state until the program requires a value for `eof` or `eoln` either explicitly or implicitly; for example, in a call to `read`. The important point to note here is that if you force Pascal to determine whether the input is at the end of file or the end of line, it is necessary for it to attempt to read from the input.

Consider the following example code:

The Pascal program, `eof_example1.p`, which shows the improper use of the `eof` function.

```
program eof_example1;

var
    i: integer;

begin
    while not eof do begin
      write('Number, please?  ');
      read(i);
      writeln('That was a ', i: 2, '.');
      writeln
    end
end. { eof_example1 }
```

At first glance, this may appear to be a correct program for requesting, reading and echoing numbers. Notice, however, that the `while` loop asks whether `eof` is true before the request is printed.

This system is forced to decide whether the input is at the end of file. It gives no messages; it simply waits for the user to type a line:

The commands to compile and execute `eof_example1.p`:

```
hostname% pc eof_example1.p
hostname% a.out
23
Number, please? That was a 23.

Number, please? ^D
standard input: Tried to read past end of file
a.out terminated by signal 5: SIGTRAP
Traceback being written to a.out.trace
Abort (core dumped)
hostname%
```

The following code avoids this problem by prompting before testing `eof`:

The Pascal program, `eof_example2.p`, which also shows the improper use of the `eof` function.

```
program eof_example2;

var
    i: integer;

begin
    write('Number, please?  ');
    while not eof do begin
      read(i);
      writeln('That was a ', i: 2, '.');
      writeln;
      write('Number, please?  ')
    end
end. { eof_example2 }
```

You must still type a line before the `while` test is completed, but the prompt asks for it. This example, however, is still not correct. To understand why, it is first necessary to know that there is an end-of-line character at the end of each line in a Pascal text file. Each time you test for the end of the file, `eof` finds the end-of-line character. Then, when `read` attempts to read a character, it skips past the end-of-line character, and finds the end of the file, which is illegal.

Thus, the modified code still results in the following error message at the end of a session:

The commands to compile and execute `eof_example2.p`:

```
hostname%  pc eof_example2.p
hostname% a.out
Number, please? 23
That was a 23.

Number, please? ^D
standard input: Tried to read past end of file
Traceback being written to a.out.trace
Abort (core dumped)
hostname%
```

The simplest way to correct the problem in this example is to use the procedure `readln` instead of `read`. `readln` also reads the end-of-line character, and `eof` finds the end of the file:

The Pascal program, `eof_example3.p`, which shows the proper use of the eof function.

```
program eof_example3;

var
    i: integer;

begin
    write('Number, please?  ');
    while not eof do begin
      readln(i);
      writeln('That was a ', i: 2, '.');
      writeln;
      write('Number, please? ')
    end
end. { eof_example3 }
```

The commands to compile and execute `eof_example3.p`

```
hostname% pc eof_example3.p
hostname% a.out
Number, please? 23
That was a 23.

Number, please? ^D
hostname%
```

In general, unless you test the end of file condition both before and after calls to `read` or `readln`, there may be inputs that cause your program to attempt to read past the end-of-file.

More About `eoln`

To have a good understanding of when `eoln` is true, it is necessary to know that in any file text there is a special character indicating end-of-line, and that, in effect, Pascal always reads one character ahead of the `read` command.

For instance, in response to `read(ch),` Pascal sets `ch` to the current input character and gets the next input character. If the current input character is the last character of the line, then the next input character from the file is the newline character, the normal operating system line separator.

When the `read` routine gets the newline character, it replaces that character by a blank (causing every line to end with a blank) and sets `eoln` to true. `eoln` is true as soon as you read the last character of the line and before you read the blank character corresponding to the end of line. Thus, it is almost always a mistake to write a program that deals with input in the following way:

This code shows the improper use of the `eoln` function.

```
read(ch);
if eoln then
     Done with line
 else
     Normal processing
```

This almost always has the effect of ignoring the last character in the line. The `read(ch)` belongs as part of the normal processing. In Pascal terms, `read(ch)` corresponds to `ch := input^; get(input)`.

This code shows the proper use of `eoln`.

```
read(ch);
if eoln then
     Done with line
else begin
     read(ch);
     Normal processing
     end
```

Given this framework, it is not hard to explain the function of a `readln` call, which is defined as follows:

```
while not eoln do
    get(input);
get(input);
```

This advances the file until the blank corresponding to the end of line is the current input symbol and then discards this blank. The next character available from `read` is the first character of the next line, if one exists.

Associating External Files with Pascal File Variables

In Pascal, most input and output routines have an argument that is a file variable. This system associates these variables with either a permanent or temporary file at compile time.

Permanent Files

Table 7-2 shows how to associate a Pascal file variable with a permanent file.

Table 7-2 Pascal File Variable with a Permanent File

Association	Description
With the `open` function.	`open` associates a permanent file with a file variable for reading or writing. `open` can also determine if a file actually exists.
With the `reset` and rewrite functions.	In Pascal, `reset` and `rewrite` take an optional second argument, a file name. If you specify the file name, the compiler opens the file and associates it with the given file variable. Any previous file associated with the file variable is lost.
With the program header.	If you call `reset` or `rewrite` with a file variable *f1*, which is bound to a file variable declared *f2* in the program header and do not specify the filename, Pascal opens a file with the same name as the variable *f2*. `reset` gives a runtime error if the file does not exist. `rewrite` creates the file if it does not exist.

Temporary Files

Table 7-3 shows how to associate a Pascal file variable with a temporary file.

Table 7-3 Pascal File Variable with a Temporary File

Association	Description
With the procedure: `rewrite(file_variable)`	`file_variable` must **not** be declared in the program statement. This procedure creates a temporary file called `#tmp.<`*suffix*`>`, where *suffix* is unique to that temporary file. When the program exits or leaves the scope in which *file_variable* is declared, the file is deleted.
With the procedure: `rewrite(output)`	The procedure creates the temporary file called `#tmp.<`*suffix*`>`, where *suffix* is unique to that temporary file. This file is not deleted after program execution.

Using `input`, `output`, *and* `errout` *Variables*

The `input`, `output`, and `errout` variables are special predefined file variables. `input` is equivalent to the SunOS 5.0 operating system standard input file `stdin`; `output` is equivalent to the SunOS 5.0 operating system standard output file `stdout`; and `errout` is equivalent to the SunOS 5.0 operating system standard error file, `stderr`.

Properties of `input`, `output`, *and* `errout` *Variables*

The `input, output,` and `errout` variables are of the type `text` and have the following special properties:

- `input`, `output`, and `errout` are optional in the program header.
- You can redirect `input`, `output`, and `errout` to files or pipe them to other programs.
- You can redefine `input`, `output`, and `errout`.
- You do not have to name `input` and `output` as explicit arguments to the `read`, `readln`, `write`, and `writeln` procedures.

- In the initial state of `input`, `eoln` is true and `eof` is false. `input`↑ is not initially defined when it is associated with `stdin` until the first `read` or `readln`. For `output`, `eoln` is initially undefined and `eof` is true.

Associating `input` *With a File Other Than* `stdin`

To associate `input` with a file other than `stdin`, call `reset(input,` *filename*). Pascal opens the SunOS 5.0 operating system file *filename* and associates it with `input`. `read` and `readln` read from that file. For example, this line opens the SunOS 5.0 operating system file `some/existing/file` and associates it with `input`:

```
reset(input,'some/existing/file');
```

If you do not supply a file name, nothing happens.

Associating `output` *With a File Other Than* `stdout`

To associate `output` with a file other than `stdout`, call `rewrite(output,` *filename*). Pascal opens the SunOS 5.0 operating system file *filename* and associates it with `output`. For example, this line associates `/home/willow/test` with `output`:

```
rewrite(output, '/home/willow/test');
```

Now, whenever you direct `write` or `writeln` to `output`, the output is sent to `/home/willow/test`. This includes the default case, when you `write` without giving a file variable.

If you call `rewrite` on `output` and you haven't associated `output` with an external file, the program creates a file with a name of the form `#tmp.`*suffix*, where *suffix* is unique to that file. Pascal does not delete this file after the program exits.

The Pascal I/O Library

Each file variable in Pascal is associated with a data structure. The data structure defines the physical SunOS 5.0 operating system file with which the variable is associated. It also contains flags that indicate whether the file variable is in an `eoln` or `eof` state.

The data structure also includes the buffer. The buffer normally contains a single component that is the same type as the type of the file. For example, a `file of char` has one character buffer, and a `file of integer` has one integer buffer.

Buffering of File Output

It is extremely inefficient for Pascal to send each character to a terminal as it generates it for output. It is even less efficient if the output is the input of another program, such as the line printer daemon, `lpr(1)`.

To gain efficiency, Pascal buffers output characters; it saves the characters in memory until the buffer is full and then outputs the entire buffer in one system interaction.

For interactive prompting to work, Pascal must print the prompt before waiting for the response. For this reason, Pascal normally prints all output that has been generated for `output` whenever one of the following occurs:

- The program calls a `writeln`.
- The program reads from the terminal.
- The program calls either the `message` or `flush` procedure.

In the following code sequence, the output integer does not print until the `writeln` occurs:

```
for i := 1 to 5 do begin
    write(i);
    Compute a lot with no output
end;
writeln;
```

Pascal performs line buffering by default. To change the default, you can compile your program with **`-b`** option. When you specify the **`-b`** option on the command line, the compiler turns on block-buffering with a block size of 1024. You may specify this option in a program comment using one of these formats:

`{$b0}` No buffering.

`{$b1}` Line buffering. This is the default.

`{$b2}` Block buffering. The block size is 1024. Any number greater than 2 (for example, `{$b5}`) is treated as `{$b2}`.

This option only has an effect in the main program. The value of the option in effect at the "`end`" statement of the main program will be used for the entire program.

I/O Error Recovery

When an I/O routine encounters an error, it normally does the following:

- Generates an error message.
- Flushes its buffers.
- Terminates with a SIGTRAP.

Although you can set up a signal handler to trap this signal, you will not be able to determine which routine called the signal or the reason it was called.

Pascal enables you to set I/O trap handlers dynamically in your program. The handler is a user-defined Pascal function.

When an I/O error occurs, Pascal runtime library checks if there is a current active I/O handler. If one does not exist, Pascal prints an error message, invokes a SIGTRAP signal, and terminates.

If a handler is present, the handler is passed the values `err_code` and `filep` as `in` parameters. The parameter `err_code` is bound to the error value that caused the I/O routine to fail. The parameter `filep` is bound to the I/O descriptor that `getfile` returned for the file in which the error occurred. If `filep` equals `nil`, no file was associated with the file variable when the error occurred.

The handler returns a boolean value. If the value is `false`, the program terminates. If the value is `true`, program execution continues with the statement immediately following the I/O routine that called the trap. The results of the I/O call remain undefined.

You can set the handler to `nil` to return it to its default state.

The scope of the active handler is determined dynamically. Pascal has restrictions as to the lexical scoping when you declare the handler. The compiler assumes that the handler is a function declared at the outer-most

level. Providing a nested function as the handler may cause unexpected results. The compiler issues a warning if it attempts to take the address of a nested procedure.

To set an I/O trap handler, you need to include the file `ioerr.h` in your Pascal source file. `ioerr.h` consists of an enumeration type of all possible I/O error values, a type declaration of an `io_handler` procedure pointer type, and an external declaration of the `set_io_handler` routine.

This file resides in the directory `/usr/lang/SC1.0/include/pascal.` (If the compiler is installed in a non-default location, change the `/usr/lang` to the location where the compiler is installed.)

The `include` file, `ioerr.h`:

```
/* Copyright 1989 Sun Microsystems, Inc. */

type
      IOerror_codes = (
              IOerr_no_error,
              IOerr_eoln_undefined,
              IOerr_read_open_for_writing,
              IOerr_write_open_for_reading,
              IOerr_bad_data_enum_read,
              IOerr_bad_data_integer_read,
              IOerr_bad_data_real_read,
              IOerr_bad_data_string_read,
              IOerr_bad_data_varying_read,
              IOerr_close_file,
              IOerr_close_null_file,
              IOerr_open_null_file,
              IOerr_create_file,
              IOerr_open_file,
              IOerr_remove_file,
              IOerr_reset_file,
              IOerr_seek_file,
              IOerr_write_file,
              IOerr_file_name_too_long,
              IOerr_file_table_overflow,
              IOerr_line_limit_exceeded,
              IOerr_overflow_integer_read,
              IOerr_inactive_file,
              IOerr_read_past_eof,
              IOerr_non_positive_format
              );


io_handler = ^function( in err_code : IOerror_codes;
                        in fileptr  : univ_ptr) :
                        boolean;

procedure set_ioerr_handler(handler : io_handler); extern c;
```

The following program illustrates how to set an I/O trap routine.

Pascal program `ioerr.p`. The program defines the I/O trap routine, `test_handler`. This routine is called each time a runtime error occurs during an I/O operation. The `#include` statement includes `ioerr.h` in the program.

```
{$w-}
program ioerr_example(output);
{ This program sets and uses an I/O trap routine. }

#include "ioerr.h"
const
    NAME = 'rmc.dat';

var
    f: text;
    IO_ERROR: IOerror_codes;
    str: array [1..10] of char := 'Testing';

function test_handler(in code: IOerror_codes;
                      in fileptr: univ_ptr): boolean;

begin
    if code = IO_ERROR then begin
      writeln('ERROR HANDLER ', code);
      test_handler := true
    end else
      test_handler := false
end; { test_handler }

begin { main program }
    set_ioerr_handler(addr(test_handler));
    { Write to an unopened file. }
    IO_ERROR := IOerr_inactive_file;
    write(f, 'This file is not open.');
    { Read a file open for writing. }
    rewrite(f, NAME);
    IO_ERROR := IOerr_read_open_for_writing;
    readln(f, str);
    remove(NAME);
    { Remove a nonexistent file. }
    IO_ERROR := IOerr_remove_file;
    remove('nonexistent.dat')
end. { ioerr_example }
```

The commands to compile and execute `ioerr.p`. When you use an I/O error recovery routine, you should compile your program with the **`-C`** option.

```
hostname% pc -C ioerr.p
hostname% a.out
ERROR HANDLER IOerr_inactive_file
ERROR HANDLER IOerr_read_open_for_writing
ERROR HANDLER IOerr_remove_file
hostname%
```

Pascal Language Reference Summary

A

This appendix is a language reference summary for Pascal.

Programs

```
<program unit> ::= <program heading> <declaration list> <program body>
```

```
<program heading> ::= 'program' <identifier>
    [ '(' <identifier list> ')' ] ';'
```

```
<program body> ::= <block> '.'
```

A

Modules

```
<module unit> ::= [ <module heading> ] <declaration list>
```

```
<module heading> ::= [ 'module' <identifier> ';' ]
```

Declarations

```
<declaration list> ::= { <declaration> }
```

```
<declaration> ::= <label declaration>
      | <variable declaration>
      | <constant declaration>
      | <type declaration>
      | <define declaration>
      | <procedure declaration>
      | <function declaration>
```

Label Declaration

```
<label declaration> ::= 'label' <label> {',' <label>} ';'
```

Variable Declaration

```
<variable declaration> ::= [ <variable scope> ] 'var' <identifier list> ':'
     [ <variable attribute> ] <type specification> [ <initialization> ] ';'
   { <identifier list> ':' [ <variable attribute> ] <type  specification>
   [ <initialization> ] ';'
```

```
<variable scope> ::= ( 'private' | 'public' )
```

```
<variable attribute> ::= ( 'static' | 'extern' | 'define' )
```

```
<initialization> ::= ':=' <initial values>
```

Constant Declaration

```
<constant declaration> ::= 'const' <identifier> '='
     <constant expression> ';' { <identifier> '='
     <constant expression> ';' }
```

Type Declaration

```
<type declaration> ::= 'type' <identifier> '=' <type> ';'
     { <identifier> '=' <type> ';' }
```

A

Define Declaration

```
<define declaration> ::= 'define' <identifier> [ <initialization> ]
{ ','<identifier> [ <initialization> ] }
```

Procedure Declaration

```
<procedure declaration> ::= <procedure heading>
<declaration list> <block>
```

```
<procedure heading> ::= [ <visibility> ] 'procedure' <identifier>
   [ '(' <parameter list> ')' ] ';' [<routine options> ]';'
```

```
<visibility> ::= ('public' | 'private')
```

```
<parameter list> ::= <parameters> { ';' <parameters> }
```

```
<parameters> ::= ( [ ('in' | 'out' | 'in out' | 'var') ]
    <identifier list> ':' [ 'univ' ] <type specification>
    | 'procedure' <identifier parameters>
    | 'function' <identifier parameters> ':' <type specification>
    | <conformant array parameters> )
```

```
<conformant array parameters> ::= [ 'var' ] [ 'packed' ] 'array'
    '[' <conformant array indicies> ']' 'of' <type identifier>
```

```
<conformant array indicies> ::= <conformant array index>
    { ';' <conformant array index> }
```

```
<conformant array index> ::= <identifier> '..' <identifier> ':'
    <type identifier>
```

```
<routine options> ::= 'options' '(' <options> { ',' <options> } ')'
    | { <options> ';' } ')'
```

```
<options> ::= ( 'forward' | 'extern' [ ( 'fortran' | 'c' ) ]
    | 'external' [ ( 'fortran' | 'c' ) ]
    | 'internal' | 'variable' | 'nonpascal' )
```

Function Declaration

```
<function declaration> ::= <function heading> <declaration list> <block>
```

```
<function heading> ::= [ <visibility> ] 'function' <identifier>
    [ '(' <parameter list> ')' ] ':' <type identifier> ';'
    [routine options> ]
```

A

Initialization

<*initialization*> ::= ':=' <*constant expression*>
| ':=' <*structure initialization*>

<*structure initialization*> ::= '[' <*element list*> ']'

<*element list*> ::= <*element*> { ',' <*element*> }

<*element*> ::= <*constant expression*>
| <*constant expression*> '..' <*constant expression*>
| <*identifer*> ':=' <*constant expression*>
| <*constant expression*> 'OF' <*constant expression*>
| '*' 'of' <*constant expression*>

<*constant expression*> ::= <*expression*>

Note – <constant expression> is an expression that can be evaluated at compile time.

Constants

<constant> ::= <character string>
| <constant identifier>
| <number>
| '+' <number>
| '-' <number>

<number> ::= <integer>
| <octal constant>
| <unsigned real constant>

<integer> ::= <digit> { <digit> }
| <base> '# ' <extended-digit> { <extended-digit> }

<base> ::= <digit> { <digit> }

<extended-digit> ::= <digit> | <letter>

<constant list> ::= <constant { ',' <constant> }

A

Types

```
<type> ::= <simple type>
      | '^' <type identifier>
      | <structured type>
      | 'packed' <structured type>
```

```
<simple type> ::= <type identifier>
      | '(' <identifier list> ')'
      | <constant> '..' <constant>
```

```
<structured type> ::= 'array' '[' <simple type list> ']' 'of' <type>
     | 'file' 'of' <type>
     | 'set' 'of' <simple type>
     | 'record' <field list> 'end'
```

```
<simple type list> ::= <simple type> { ',' <simple type> }
```

Record Types

```
<field list> ::= <fixed part>
      | <variant part>
      | <fixed part> <variant part>
```

```
<fixed part> ::= [ <field> ';' ]
```

<*field*> ::= <*empty*>
| <*identifier list*> ':' <*type*>

<*variant part*> ::= <*empty*>
| 'case' <*type identifier*> 'of' <*variant list*>
| 'case' <*identifier*> ':' <*type identifier*> 'of' <*variant list*>

<*variant list*> ::= <*variant*>
| { ';' <*variant*> }

<*variant*> ::= <*empty*>
| <*constant list*> ':' '(' <*field list*> ')'

Statements

<*block*> ::= 'begin' <*statement list*> 'end'

<*statement list*> ::= { <*statement*> ';' }

```
<statement> ::= <label> ':' <statement>
      | <block>
      | <assignment statement>
      | <assert statement>
      | <call statement>
      | <case statement>
      | <exit statement>
      | <for statement>
      | <goto statement>
      | <if statement>
      | <next statement>
      | <null statement>
      | <return statement>
      | <while statement>
      | <with statement>
      | <write statement>
```

```
<assignment statement> ::= <variable> ':=' <expression>
```

```
<assert statement> ::= 'assert' '(' <boolean expression> ')'
```

```
<call statement> ::= <procedure identifier> [ '(' <expression>
      { ',' <expression> } ')' ]
      | <qualified variable> '^' [ '(' <expression>
      { ',' <expression> } ')' ]
```

```
<case statement>  ::= 'case' <expression> 'of'
      { <case values> ':' <statement> ';' }
      [ 'otherwise' <statement list> ]
      'end'
```

<*case values*> ::= <*case range*> { ',' <*case range*> }

<*case range*> ::= <*constant expression*>
| <*constant expression*> '..' <*constant expression*>

<*exit statement*> ::= 'exit'

<*for statement*> ::= 'for' <*control variable*> ':=' <*initial value*
('to' | 'downto') <*final value*> 'do' <*statement*>

<*goto statement*> ::= 'goto' <*label*>

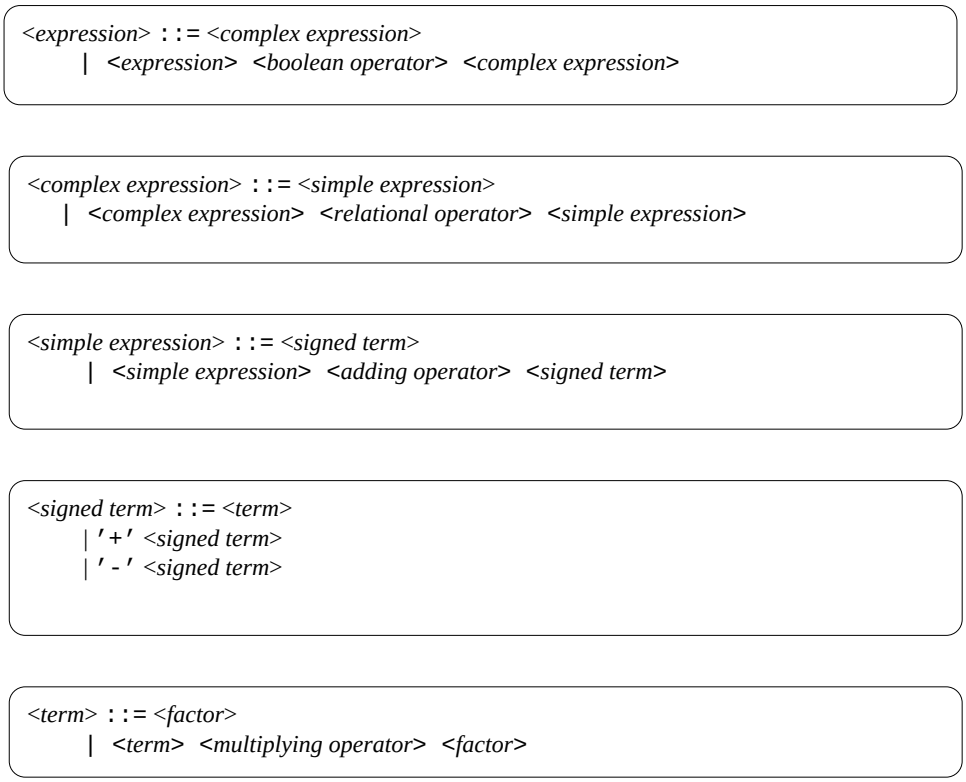
<*label*> ::= <*unsigned integer*> | <*identifier*>

<*if statement*> ::= 'if' <*boolean expression*> 'then' <*statement*>
['else' <*statement*>]

<*next statement*> ::= 'next'


<null statement> ::= ';'
<return statement> ::= 'return'
<while statement> ::= 'while' <boolean expression> 'do' <statement>
<with statement> ::= 'with' <record variable list> 'do' <statement>
<record variable list> ::= <record variable> { ',' <record variable> }
<record variable> ::= <variable> [':' <with_identifier>]
<with_identifier> ::= <identifier>
<write statement> ::= ('write' | 'writeln')
['(' <write parameters> ')']

Expressions



```
<factor> ::= 'nil'
      | <character string>
      | <unsigned integer>
      | <octal constant>
      | <unsigned real constant>
      | <variable>
      | <function identifier> '(' <actual parameter list> ')'
      | '(' <expression> ')'
      | 'not' <factor>
      | '~' <factor>
      | '[' <set element list> ']'
      | '[' ']'
      | <type-cast function>
```

```
<type-cast function> ::= <type identifier> '(' <expression> ')'
```

```
<set element list> ::= <set element>
      | { ',' <set element> }
```

```
<set element> ::= <expression> [.. <expression> ]
```

Variables

```
<variable> ::= <identifier>
| <qualified variable>
```

```
<qualified variable> ::= <array identifier> [ <expression list> ]
      | <qualified variable> [ <expression list> ]
      | <record identifier> '.' <field identifier>
      | <qualified variable> '.' <field identifier>
      | <pointer identifier> '^'
      | <qualified variable> '^'
```

Write

```
<write parameter> ::= <expression>
      | <expression> ':' <expression>
      | <expression> ':' <expression> ':' <expression>
      | <expression> <write base>
      | <expression> ':' <expression> <write base>
```

```
<expression list> ::= <expression>
      | { ',' <expression> }
```

```
<actual parameter list> ::= <actual parameter>
      | { ',' <actual parameter> }
```

```
<write base> ::= 'oct' | 'hex'
```

Operators

```
<relational operator> ::= '=' | '<' | '>' | '<>' | '<='
      | '>=' | 'in'
```

A

<*adding operator*> ::= ’+’ | ’-’ | ’or’ | ’|’ | ’!’

<*multiplying operator*> ::= ’*’ | ’/’ | ’div’ | ’mod’ | ’and’
| ’&’

<*boolean operator*> ::= ’or’ ’else’ | ’and’ ’then’

Lexicon

<*constant identifier*> ::= <*identifier*>

<*type identifier*> ::= <*identifier*>

<*var identifier*> ::= <*identifier*>

<*array identifier*> ::= <*identifier*>

<*pointer identifier*> ::= <*identifier*>

<*record identifier*> ::= <*identifier*>

<*field identifier*> ::= <*identifier*>

<*procedure identifier*> ::= <*identifier*>

<*function identifier*> ::= <*identifier*>

<*identifier*> ::= <*letter*> | '_' | '$' { <*letter*> | <*digit*>
| '_' | '$' }

```
<letter> ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h'
    | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q'
    | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
    | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I'
    | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
    | 'S' | 'T'| 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
```

```
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
      | '8' | '9'
```

```
<unsigned integer> ::= <digit> { <digit> }
```

```
<signed integer> ::= <unsigned integer>
      | '+' <unsigned integer>
      | '-' <unsigned integer>
```

```
<unsigned real constant> ::= <unsigned integer> '.' <fractional part>
      | <unsigned integer> '.' <fractional part> 'e' <scale factor>
      | <unsigned integer> '.' <fractional part> 'E' <scale factor>
      | <unsigned integer> 'e' <scale factor>
      | <unsigned integer> 'E' <scale factor>
```

```
<fractional part> ::= <digit> { <digit> }
```

```
<scale factor> ::= <signed integer>
```

```
<octal constant> ::= <octal digit> { <octal digit> } 'b'
      | <octal digit> { <octal digit> } 'B'
```

```
<octal digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6'
      | '7'
```

<*character string*> ::= ''' <*string element*> { <*string element*> } '''

<*string element*> ::= <*apostrophe image*>
| <*any character except apostrophe or newline*>

<*apostrophe image*> ::= ''''

Miscellaneous

<*identifier list*> ::= <*identifier*> { ',' <*identifier*> }

<*empty*> ::=

<*procedure pointer*> ::= '^' <*procedure heading*>
| <*type identifier*>

<*function pointer*> ::= '^' <*function heading*>
| <*type identifier*>

A

Overview of Pascal Extensions B

This Appendix gives an overview of the Pascal extensions to ISO / ANSI standard Pascal. The extensions marked with one bullet (•) are Sun Pascal 1.1 extensions. Those marked with two bullets (••) are Sun Pascal 2.1 extensions and SPARCompiler 3.0 extensions.

Lexical Elements

Pascal supports the following extensions to the lexical elements of standard Pascal:

- Upper and lower case-sensitive.
- The special symbols ~, &, |, !, #, and %.
- The reserved words `external`, `otherwise`, `private`, `public`, and `univ`.
- •• The reserved words `define`, `extern`, `module`, and `static`.
- The identifiers in Table B-1.
- •• The identifiers in Table B-2.
- An `underscore (_)` and `dollar_sign($)` in identifier names.
- The comment delimiters `/* */`. in addition to the standard `(* *)` and `{ }`
- •• The comment delimiters `" "` .

B

Table B-1 Sun Pascal 1.1 Nonstandard Identifiers

Nonstandard Identifiers in Sun Pascal 1.1				
FALSE	card	land	message	seed
TRUE	clock	length	minchar	shortreal
alfa	date	linelimit	minint	sizeof
argc	exit	lnot	next	
argv	expo	longreal	null	
asl	flush	lor	pack	
asr	getfile	lsl	random	
assert	halt	lsr	remove	
bell	intset	maxchar	return	

Table B-2 Sun Pascal 2.1/SPARCompiler Pascal 3.0 Nonstandard Identifiers

Nonstandard Identifiers in Sun Pascal 2.1 and SPARCompiler Pascal 3.0		
addr	in_range	min
arshft	index	open
close	integer16	rshft
discard	integer32	single
double	lastof	substr
firstof	lshft	trim
getenv	max	univ_ptr

Data Types

Pascal supports these extensions to the standard Pascal data types:

- The `real` data types `shortreal` and `longreal`.
- The `real` data types `single` and `double`.
- A `real` constant without a digit after the decimal point.
- The integer data types `integer16` and `integer32`.
- An integer constant in another base.

- Character constants `minchar`, `maxchar`, `bell`, and `tab`.
- Fixed- and variable-length character strings.
- Array initialization using a default upper bound or a repeat count.
- A set of type `intset`, which contains the elements 0 through 127.
- A pointer type to procedures and functions.
- A universal pointer type that holds a pointer to a variable of any data type.

Statements

Pascal extends the standard definition of statements as follows:

- The `and then` and `or else` operators in the `if` statement.
- The `assert` statement.
- The `otherwise` statement in a `case` statement.
- Constant ranges in a `case` statement.
- The `exit` statement in a `for`, `while`, or `repeat` loop.
- The `next` statement in a `for`, `while`, or `repeat` loop.
- An identifier as the target of a `goto` statement.
- The `return` statement in a procedure or function.
- An alternative format of the `with` statement.

Assignments and Operators

Pascal supports these extensions to standard Pascal operators:

- The bitwise operators ~(`not`), &(`and`), |(`or`), and !(or).
- The `boolean` operators `and then` and `or else`.
- The relational operators on sets.
- The equality (=) and inequality <>) (operators on records and arrays.
- The concatenation operator, the plus sign (+), on any combination of fixed- and variable-length strings.

B

Headings and Declarations

Pascal supplies the following extensions to the standard program heading and declarations:

- Identifiers as labels.
- A constant equal to a set expression.
- `public` and `private` variable declarations.
- The `static`, `extern`, and `define` variable attributes.
- Real, integer, boolean, character, set, record, array, and pointer variable initialization in the `var` declaration.
- The `define` declaration.
- The `label`, `const`, `type`, `var`, and `define` declaration in any order and any number of times.

Procedures and Functions

Pascal supports the following extensions to the standard Pascal definition of procedures and functions:

- `public` and `private` procedure and function declarations.
- The `in`, `in out`, and `out` parameter types.
- The `univ` keyword parameter type.
- The `extern`, `external`, `internal`, `variable`, and `nonpascal` routine options.

Built-in Routines

Pascal supports the following nonstandard built-in routines:

- The `addr` function, which returns the address of a specified variable.
- The `argc` function, which returns the number of arguments passed to the program.
- The `argv` procedure, which assigns the specified program argument to a string variable.

- The `arshft` function, which does an arithmetic right shift of an integer.
- The `asl` function, which does an arithmetic left shift of an integer.
- The `asr` function, which is identical to `arshft`.
- The `card` function, which returns the cardinality of a set.
- The `clock` function, which returns the user time used by the process.
- The `close` procedure, which closes the specified file.
- The `date` procedure, which fetches the current date.
- The `discard` procedure, which explicitly discards the return value of a function.
- The `expo` function, which calculates the exponent of a specified variable.
- The `firstof` function, which returns the first possible value of a type or variable.
- The `flush` procedure, which writes the output buffered for the specified Pascal file into the associated SunOS 5.0 operating system file.
- The `getenv` function, which returns the value associated with an environment name.
- The `getfile` function, which returns a pointer to the C standard I/O descriptor associated with the specified Pascal file.
- The `halt` procedure, which terminates program execution.
- The `index` function, which returns the position of the first occurrence of a string or character within another string.
- The `in_range` function, which determines whether a specified value is in the defined integer subrange.
- The `land` function, which returns the bitwise `and` of two integer values.
- The `lastof` function, which returns the last possible value of a type or variable.
- The `length` function, which returns the length of a string.
- The `linelimit` function, which terminates execution of a program after a specified number of lines have been written into a text file.
- The `lnot` function, which returns the bitwise `not` of an integer value.

- The `lor` function, which returns the inclusive `or` of two integer values.
- The `lshft` function, which does a logical left shift of an integer.
- The `lsl` function, which is identical to `lshft`.
- The `lsr` function, which is identical to `rshft`.
- The `max` function, which returns the larger of two expressions.
- The `message` procedure, which writes the specified information to `stderr`.
- The `min` function, which returns the smaller of two expressions.
- The `null` procedure, which performs no operation.
- The `open` procedure, which associates an external file with a file variable.
- The `random` function, which generates a random number between 0.0 and 1.0.
- The `read` and `readln` procedures, which read in boolean variables, fixed- and variable-length strings, enumerated types, and pointers from the standard input.
- The `remove` procedure, which removes the specified file.
- The `reset` and `rewrite` procedures, which accept an optional second argument, a SunOS 5.0 operating system file name.
- The `rshft` function, which does a logical right shift of an integer.
- The `seed` function, which reseeds the random number generator.
- The `sizeof` function, which returns the size of a specified type.
- The `stlimit` procedure, which terminates program execution if a specified number of statements have been executed in the current loop.
- The `substr` function, which extracts a substring from a string.
- The `sysclock` function, which returns the system time used by the process.
- The `time` procedure, which retrieves the current time.
- The `trim` function, which removes the trailing blanks in a character string.
- The *type_transfer* function, which permits you to change the data type of a variable or expression.
- The `wallclock` function, which returns the elapsed number of seconds since 00:00:00 GMT January 1, 1970.

- The `write` and `writeln` procedures, which output enumerated type values to the standard output and allow output expressions in octal or hexadecimal.
- The `write` and `writeln` procedures, which allow negative field widths.
- The `xor` function, which returns the exclusive `or` of two integer values.

Input and Output

Pascal supports the following extensions to standard Pascal input and output:

- Associate a Pascal file with either a permanent or temporary SunOS 5.0 operating system file.
- The special predefined file variables, `input` and `output` need not be specified in the program statement.
- The special predefined file variable, `errout`.
- An I/O error recovery mechanism.

Program Compilation

Pascal supports the following extensions to program compilation:

- Share variable, procedure, and function declarations across multiple units using `include` files.
- Share variable, procedure, and function declarations across multiple units using multiple declarations.
- Share variable, procedure, and function declarations across multiple units using the `extern` and `define` variable declarations.
- Share variable, procedure, and function declarations between units of different languages using the `extern` and `external` routine options.

B

Pascal and DOMAIN Pascal C

This Appendix describes the differences between Pascal and Apollo DOMAIN Pascal and how the `-xl` option can be used to get around most of these differences.

The `-xl` *Option*

The `-xl` option to the `pc` command makes the language accepted by the Pascal compiler similar to DOMAIN Pascal. Table C-1 lists the differences in your program when you compile it with and without `-xl`.

Table C-1 Differences Between Programs Compiled With and Without `-xl`

With `-xl`	**Without `-xl`**
The default `integer` size is 16 bits.	The default is 32 bits.
The default `real` size is 32 bits.	The default is 64 bits.
The default enumerated type size is 16 bits.	The default is either 8- or 16-bits, depending on the number of elements in the enumerated set.
The source file is run through the preprocessor `cppas` before it is processed by the compiler.	The source file is run through the preprocessor `cpp`.

Table C-1 Differences Between Programs Compiled With and Without `-xl`

With `-xl`	Without `-xl`
Pascal supports `nonpascal` as a routine option.	`Note that nonpascal` is not supported.
The `-L` option, which maps all identifiers to lower `case`, is on by default.	`-L` is off by default.
If the value of the expression in a `case` statement does not match any of the `case` values, the program falls through and does not generate an error. The program continues execution in the statement immediately following the `case` statement.	The compiler generates an error and halts.
The writing of enumerated and boolean variables defaults to upper case and 15-character width format.	Enumerated variables default to the length of the type. Boolean variables default to the length of true or false
Integer or real constant literals that overflow implementation limits do not cause an error. The resulting action is undefined.	An error is generated.
No warning is generated when the argument to the `addr` function is a local or private variable.	A warning is generated.
Top-level variables, procedures, and functions in programs default to private.	Variables, procedures, and functions in programs default to public.
Top-level variables in modules default to private.	Variables in modules default to public.
Modules compiled with `-xl` are *not* compatible with modules compiled without `-xl`.	You should not link these two types of modules together.

DOMAIN Pascal Features Accepted But Ignored

Pascal accepts these DOMAIN Pascal features, but otherwise ignores them, with a warning message as appropriate:

- The `volatile`, `device`, and `address` extensions for attributes of variables and types.
- Routine attribute lists.
- The routine options `abnormal`, `nosave`, `noreturn`, `val_param`, and `d0_return`, `a0_return`, and `c_param`.

DOMAIN Pascal Features Not Supported

Pascal does not support these features of DOMAIN Pascal:

- Alignment specific to the DN10000 in DOMAIN Pascal SR10.
- Allocation of variables into named sections.
- Calls to the DOMAIN system libraries.
- Compiler directives inside comments.
- The functions `append`, `ctop`, `find`, `ptoc`, `replace`, and `undefined`.
- Special characters embedded in string literals.
- The system programming routines `disable`, `enable`, and `set_sr`.

C

Implementation Restrictions D

This Appendix describes the Pascal features that are implementation-defined.

Identifiers

Pascal restricts the maximum length of an identifier to 1024 characters. All characters are significant.

Be aware that identifiers in a nested procedure are concatenated with the identifier of the containing procedure. Thus, an identifier in a deeply nested procedure may become several hundred characters when concatenated and may cause problems with the compiler. Pascal generates an error when this situation occurs.

Data Types

This section describes the restrictions Pascal places on the following data types:

- real
- integer
- character
- record
- array
- set
- alignment

D

Real

Table D-1 lists the minimum and maximum values Pascal assigns to the real data types `single` and `double`.

Table D-1 Values for `single` and `double`

Type	Bits	Maximum Value	Minimum Value
`single`	32 bits	3.402823e+38	1.401298e-45
`double`	64 bits	1.79769313486231470e+308	4.94065645841246544e-324

Integer

The value Pascal assigns to the integer constants `maxint` and `minint` depends on whether or not you compile your program with the `–xl` option, as shown in Table D-2.

Table D-2 `maxint` and `minint`

Option	`maxint`	`minint`
`-xl off`	2147483647	-2147483648
`-xl on`	32767	-32768

Character

Pascal defines the maximum range of characters as 0 to 255.

Record

Pascal restricts the maximum size of a record to 2147483647 bytes.

Array

Pascal restricts the maximum size of an array to 2147483647 bytes.

Set

Pascal restricts the maximum size of a set to 32767 elements.

Alignment

The size and alignment of data types depends on whether or not you compile your program with the `-xl` option. Table D-3 shows the representation of data types without `-xl`, and Table D-4 shows the representation with `-xl`.

Table D-3 Internal Representation of Data Types Without `-xl`

Data Type	Size	Alignment
`integer`	4 bytes	4 bytes
`integer16`	2 bytes	2 bytes
`integer32`	4 bytes	4 bytes
`real`	8 bytes	8 bytes
`single`	4 bytes	4 bytes
`shortreal`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`longreal`	8 bytes	8 bytes
`boolean`	1 byte	1 byte
`char`	1 byte	1 byte
enumerated	1 or 2 bytes depending on the number of elements in the enumerated set.	1 or 2 bytes
subrange	1,2, or 4 bytes.	1, 2, or 4 bytes
record	The space required for a particular field in a record is dependent upon the base type of that field.	4 bytes
array	Requires the same space required by the base type of the array.	Same as element type
set	Pascal implementssets as a bit vector, with one bit representing each element of a set. The size is determined by the size of the ordinal value of maximal element of the set plus one. The size is a minimum of 2 bytes and always in 2-byte multiples.	4 bytes
pointer	4 bytes	4 bytes

Table D-4 Internal Representation of Data Types With `-xl`

Data Type	Size	Alignment
`integer`	2 bytes	4 bytes
`integer16`	2 bytes	2 bytes
`integer32`	4 bytes	4 bytes
`real`	4 bytes	8 bytes
`single`	4 bytes	4 bytes
`shortreal`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`longreal`	8 bytes	8 bytes
`boolean`	1 byte	1 byte
`char`	1 byte	1 byte
enumerated	2 bytes	2 bytes
subrange	2 or 4 bytes	2 or 4 bytes
record	The space required for a particular field in a record is dependent upon the base type of that field.	4 bytes
array	Needs the same space required by the base type of the array.	Same as element type
set	Pascal implements sets as a bit vector, with one bit representing each element of a set. The size is determined by the size of the ordinal value of maximal element of the set plus one. The size is a minimum of 2 bytes and always in 2-byte multiples.	4 bytes
pointer	4 bytes	4 bytes

D

Nested Routines

Pascal allows a maximum of 20 levels of procedure and function nesting.

Default Field Widths

The write and writeln statements assume the default values in Table D-5 if you do not specify the minimum field length of a parameter.

Table D-5 Default Filed Widths

Data Type	Default Width without -xl option	Default Width with -xl option
`array of char`	Declared length of the array	Declared length of the array
`boolean`	length of true or false	15
`char`	1	1
`double`	21	21
enumerated	length of type	15
hexadecimal	10	10
`integer`	10	10
`integer16`	10	10
`integer32`	10	10
`longreal`	21	21
octal	10	10
`real`	21	13
`shortreal`	13	13
`single`	13	13
string constant	Number of characters in the string	Number of characters in the string
variable-length string	Current length of the string	Current length of the string

Incompatibilities Between Sun Pascal 1.1 and 3.0.2

E

This appendix lists the incompatibilities between Sun Pascal Release 1.1 and Pascal 3.0.2.

Reserved Words

The 1.1 nonstandard reserved words differ from the 3.0.2 nonstandard reserved words as follows:

Release 1.1

```
assert
otherwise
out
sizeof
univ
```

Release 3.0.2

```
define    private
extern    public
external  static
module    univ
otherwise
```

The only nonstandard reserved words 1.1 and 3.0.2 have in common are `otherwise` and `univ`. In 3.0.2, `assert`, `out`, and `sizeof` are predeclared identifiers.

E

Records

Pascal 1.1 ignores records defined as `packed`. In Pascal 3.0.2, records defined as `packed` affect how the compiler allocates data space.

Sets

The `set` data type is represented differently in 1.1 and 3.0.2.

Parameters

The parameters `in`, `out`, and `in out` are implemented differently in 1.1 and 3.0.2. However, these parameters perform the same functions in both releases.

Runtime Library

The internal names of the Pascal 1.1 runtime library routines differ from those of the Pascal 3.0.2 library routines. In 3.0.2, all Pascal library names begin with `__PC0__`, except `__argv`. The names were changed to prevent name clashes with user-defined variables, types, and routines.

Separate Compilation

Modules compiled with 1.1 are *not* compatible with 3.0.2 because of changes to `libpc` and changes in the data representation to sets.

The rules of separate compilation have changed from release 1.1 to 3.0.2 as shown in Table E-1.

Table E-1 Separate Compilation in Pascal 1.1 and 3.0.2

Pascal 1.1	Pascal 3.0.2
Uses `include` files to share common variables across separately compiled modules	Supports three methods for sharing variable declarations across separately compiled modules: 1. `include` files. 2. Multiple declarations, one per file. This is similar to C. The `extern` and `define` variable declarations. 3. The `extern` and `define` variable declarations. These methods are not mutually exclusive. For example, you can declare a variable as either `extern` or `define` in an `include` file.
Uses an `include` file to declare external functions and procedures.	May declare external functions and procedures outside the `include` file, using the `extern` and `external` routine options.
Enforces strict type checking across separate source files.	Does not enforce strict type checking across separate source files.
Enforces the rules of separate compilation through a special pre-linker, `pc3`.	To enforce the compilation rules of 1.1, compile your program with the `-G` option.

`case` *Statement*

In Sun Pascal 1.1, the '`otherwise`' clause in a `case` statement could occur anywhere in the `case` statement and allowed a single statement following the '`otherwise`.' In SPARCompiler Pascal 3.0.2, the '`otherwise`' clause must occur at the end of the `case` statement and allows multiple statements following the '`otherwise`.'

E

Glossary

This glossary defines terms specific to Pascal and the SPARC workstation.

Special Characters

|

Bitwise `or` operator.

~

Bitwise `not` operator.

!

Bitwise or operator.

#

Specifies an integer value in a base other than 10; includes a file in your program; indicates a preprocessor command.

%

Used with the –xl option for special cppas directives.

&

Bitwise `and` operator.

A

adb

An interactive, general-purpose, assembly-level debugger.

addr

A built-in function that returns the address of a specified variable.

`alfa`

An `array of char` 10 characters long.

`and then`

An operator similar to the standard `and` operator. The difference is that `and then` enforces left-to-right evaluation and evaluates the right operand only if the left-operand is true.

argc

A built-in function that returns the number of arguments passed to the program.

`argv`

A built-in procedure that assigns the specified program argument to a string variable.

arshft

A built-in function that does an arithmetic right shift of an integer value.

`asl`

A built-in function that does an arithmetic left shift of an integer value.

asr

A built-in function that does an arithmetic right shift of an integer value. Same as `arshft`.

assert

A statement which causes a boolean expression to be evaluated and aborts the program if false, if the –C option is specified.

B

bell

A predeclared character constant equal to `char(7)`, which makes the terminal beep.

bit operator

An operator that calculates `and`, `not`, or `or` on a bit-by-bit basis.

block buffering

Output buffering with a block size of 1024.

C

`card`
: A built-in function that returns the number of elements of a set variable.

case-sensitive
: Treating lower case and upper case characters as two kinds of characters with separate functions.

clock
: A built-in function that returns the user time consumed by the process.

close
: A built-in procedure that closes a file.

command line
: A string of characters beginning with a command followed by arguments, which are not necessarily required, including options, filenames, and other expressions.

compatibility
: Characteristic of computer equipment where one machine can use the same programs as another machine without conversion or code modification.

compiler directive
: A percent sign (%) followed by a name indicating an action for the `cppas` preprocessor to take. Programs that contain compiler directives must be compiled with the `–xl` option.

conditional variable
: A variable, either defined or undefined, handled by the `cppas` preprocessor. A conditional variable is defined when it appears in a `%var` directive. Programs that contain conditional variables must be compiled with the `–xl` option.

`%config`
: A compiler directive that is a special predefined conditional variable with a value of either true or false. Programs that contain the `%config` directive must be compiled with the `–xl` option.

`cpp`
: The C language preprocessor.

`cppas`

The preprocessor that handles the Pascal conditional variables and compiler directives when the `-xl` option is specified.

D

`date`

A built-in procedure that fetches the current date (as assigned when the operating system was initialized) and assigns it to a string variable.

`dbx`

A symbolic debugger that understands Pascal, Modula-2, C, and FORTRAN programs.

`debugger`

A window- and mouse-based version of the symbolic debugger.

`%debug`

A compiler directive that works with the `-cond` compiler option. `-cond` tells `pc` to compile the lines in your program that begin with `%debug`. Programs that contain the `%debug` directive must be compiled with the `–xl` option.

`define` attribute

An attribute used to declare a variable that is allocated in the current module and whose scope is public.

`define` declaration

A declaration used to declare a variable that is allocated in the current module and whose scope is public.

`discard`

A built-in procedure that throws away the value a function returns.

`double`

A real data type that represents a 64-bit floating-point number. Same as `longreal`.

E

%else

A compiler directive that provides an alternative action to the %if directive. If the expression in %if is false, the compiler skips over the %then part and executes the %else part instead. Programs that contain compiler directives must be compiled with the –xl option.

%elseif

A compiler directive that provides another alternative action to the %if directive. If the expression in %if is false, the compiler skips over the %then part and executes the %elseif part instead. Programs that contain compiler directives must be compiled with the –xl option.

%elseifdef

A compiler directive that provides an alternative action to the %ifdef directive. If the expression in %ifdef is false, the compiler skips over the %then part and executes the %elseifdef part instead. Programs that contain compiler directives must be compiled with the –xl option.

%enable

A compiler directive that sets a conditional variable to true. Programs that contain the %enable directive must be compiled with the –xl option.

%endif

A compiler directive that indicates the end of the %if or %ifdef directive. Programs that contain compiler directives must be compiled with the –xl option.

%error

A compiler directive that prints a string on the standard output and treats it as an error. Programs that contain the %error directive must be compiled with the –xl option.

errout

A special predefined file variable equivalent to the SunOS 5.0 operating system standard error file, stderr.

exit

A statement used in a for, while, or repeat loop to transfer program control to the first statement after the loop.

`%exit`

A compiler directive that tells the compiler to stop processing the current Pascal source file. Programs that contain the `%exit` directive must be compiled with the `-xl` option.

expo

A built-in function that calculates the integer-valued exponent of a specified number.

`extern` **attribute**

An attribute used to declare a variable that is not allocated in the current program or module unit, but is a reference to a variable allocated in another unit.

`extern` **option**

A procedure and function option that indicates the procedure or function is defined in a separate program or module unit, and possibly in a different source language. Same as `external`.

external

A procedure and function option that indicates the procedure or function is defined in a separate program or module unit, and possibly in a different source language. Same as `extern`.

F

`firstof`

A built-in function that returns the first possible value of a type or variable.

flush

A built-in procedure that writes the output buffered for the specified Pascal file into the associated SunOS 5.0 operating system file.

G

`getenv`

A built-in function that returns the value associated with an environment name.

`getfile`

A built-in function that returns a pointer to the C standard I/O descriptor associated with a Pascal file.

H

halt

A built-in procedure that terminates program execution.

hostname

The name of the machine.

I

%if

When the compiler encounters an `%if` *expression* `%then` directive, it evaluates *expression*. If *expression* is true, the compiler executes the statements after `%then`. If *expression* is false, the compiler skips over `%then`. Programs that contain compiler directives must be compiled with the `-xl` option.

`%ifdef`

A compiler directive that allows you to determine whether or not you previously defined a conditional variable in a `%var` directive. Programs that contain compiler directives must be compiled with the `-xl` option.

I/O handler

A Pascal function passed the values `err_code` and `filep` when an I/O error occurs. The handler returns `false` to terminate the program or `true` to continue program execution.

`in`

A parameter type indicating the parameter can only pass a value into a procedure or function.

`in out`

A parameter type indicating the parameter can both take in values and pass them back out.

`in_range`

A built-in function that checks if a value is in a defined subrange.

%include

A compiler directive that tells `cppas` to insert the lines from the specified file in the input stream. Programs that contain the `%include` directive must be compiled with the `-xl` option.

include file

A file that is inserted into a source file with the `%include` or `#include` directives.

independent compilation

The compilation of a multi-file program in which the compiler does not check for the consistent use of global names and types across different units.

index

A built-in function that returns the position of the first occurrence of a string or character in another string.

input

A special predefined file variable equivalent to the SunOS 5.0 operating system standard input file, `stdin`.

integer16

An integer data type that represents a 16-bit value.

integer32

An integer data type that represents a 32-bit value.

internal

A procedure and function option that makes the procedure or function local to a module.

intset

A predefined set of [0..127].

L

land

A built-in function that returns the bitwise `and` of two integers.

lastof

A built-in function that returns the last possible value of a type or variable.

length

A built-in function that returns the length of a string.

line buffering

The buffering of output line-by-line.

linelimit

A built-in procedure that terminates execution of a program after a specified number of lines have been written into a text file.

%list

A compiler directive that enables a listing of the program. Programs that contain the `%list` directive must be compiled with the `–xl` option.

lnot

A built-in function that returns the bitwise `not` of an integer value.

longreal

A real data type that represents a 64-bit floating-point number. Same as `double`.

lor

A built-in function that returns the inclusive `or` of two integer values.

lshft

A built-in function that does a logical left shift of an integer value.

lsl

A built-in function that does a logical left shift of an integer value. Same as `lshft`.

lsr

A built-in function that does a logical right shift of an integer value. Same as `rshft`.

M

max

A built-in function that evaluates two scalar expression and returns the larger one.

maxchar

A predeclared character constant equal to `char(255)`.

maxint

An integer constant that represents the 16-bit value 32,767 when you compile your program with the `–xl` option; otherwise, `maxint` represents the 32-bit value 2,147,483,647.

message

A built-in procedure that writes the specified information on `stderr` (usually the terminal).

`min`

A built-in function that evaluates two scalar expressions and returns the smaller one.

minchar

A predeclared character constant equal to `char(0)`.

`minint`

An integer constant that represents the 16-bit value -32,768 when you compile your program with the `–xl` option; otherwise, `minint` represents the 32-bit value -2,147,483,648.

module heading

A heading that contains the reserved word `module` followed by an identifier. For example, `module sum;` is a legal module heading.

module unit

A source program that does not have a program header.

N

next

A statement used in a `for`, `while`, or `repeat` loop to skip to the next iteration of the current loop.

`%nolist`

A compiler directive that disables the program listing. Programs that contain the `%nolist` directive must be compiled with the `–xl` option.

`nonpascal`

A procedure and function option that declares non-Pascal routines when you are porting Apollo DOMAIN programs written in DOMAIN Pascal, FORTRAN, C, and C++.

`null`

A built-in procedure that performs no operation.

null string

A string that does not have a value.

O

object file

A computer file containing the assembled machine code produced by the assembler.

on-line Man Pages

Also known as man pages, the on-line manual pages contain documentation of SunOS 5.0 operating system commands. They permit you to view the documentation on your screen.

open

A built-in procedure that associates an external file with a file variable.

or else

An operator similar to the standard `or` operator. The difference is that `or else` enforces left-to-right evaluation and evaluates the right operand only if the left-operand is false.

otherwise

A Pascal extension to the standard Pascal `case` statement. If the value of the case selector is not in the case label list, Pascal executes the statements in the `otherwise` clause.

out

A parameter indicating that the parameter will be used to pass values out of the routine.

`output`

A special predefined file variable equivalent to the SunOS 5.0 operating system standard output file `stdout`.

P

pc

The Pascal compiler. If given an argument file ending with `.p` or `.pas`, `pc` compiles the file and leaves the result in an executable file, called `a.out` by default.

preprocessor

A program or device that prepares data for further processing.

private

A variable, procedure, and function declaration that restricts its accessibility to the current compilation unit.

procedure and function pointer

A pointer that has the address of a procedure or function as its value.

program unit

The source program, which is made up of the program header, declaration list, and program body.

public

A variable, procedure, or function declaration that is visible across multiple programs and modules.

R

random

A built-in function that generates a random number between 0.0 and 1.0.

remove

A built-in procedure that removes the specified SunOS 5.0 operating system file.

return

A statement used in a procedure or function to prematurely end the procedure or function.

routine

Either a procedure or a function.

rshft

A built-in function that does a logical right shift of an integer value.

S

seed

A built-in function that reseeds the random number generator.

separate compilation

The compilation of a multi-file program in which the compiler checks for the consistent use of global names and types across different units.

shortreal

A real data type that represents a 32-bit floating point number. Same as `single`.

single

A real data type that represents a 32-bit floating point number. Same as `shortreal`.

sizeof

A built-in function that returns the number of bytes the program uses to store a data object.

%slibrary

A compiler directive that tells `cppas` to insert the lines from the specified file in the input stream. Same as `%include`. Programs that contain the `%slibrary` directive must be compiled with the `–xl` option.

static

A variable attribute that declares the variable private in scope.

stderr

The standard operating system error file.

stdin

The standard operating system input file.

stdout

The standard operating system output file.

stlimit

A built-in procedure that terminates program execution if a specified number of statements have been executed in the current loop.

string

An `array of char` 80 characters long.

substr

A built-in function that extracts a substring from a string.

SunOS

The Sun Operating System is a merged version of the Berkeley 4.2/4.3 UNIX and System V UNIX operating systems.

sysclock

A built-in function that returns the system time consumed by the process.

T

tab

A predeclared character constant equal to `char(9)`, which makes a tab character.

Text Editor

A menu-driven, window-based editor that allows you to edit standard text.

time

A built-in procedure that retrieves the current time.

trim

A built-in function that removes trailing blanks in a character string.

type transfer function

A built-in function that changes the data type of a variable, constant, or expression.

U

unit

Either a program or a module.

`univ`

A modifier used before data types in formal parameter lists to turn off type-checking for that parameter.

`univ_ptr`

See universal pointer.

universal pointer

A pointer used to compare a pointer of one type to another or to assign a pointer of one type to another.

unsigned integer

An integer data type, without a specific value, which must be defined by a subrange declaration that indicates the lower and upper limits of the data type.

V

`%var`

A compiler directive that defines conditional variables for the preprocessor. Programs that contain the `%var` directive must be compiled with the `-xl` option.

variable attribute

An attribute that determines how to allocate the variable. Variable attributes include `static`, `extern`, and `define`.

variable initialization

The initialization of a real, integer, boolean, character, set, record, array, or pointer variable in the `var` declaration of the program.

`variable` routine option

A routine option that allows you to pass a routine a smaller number of actual arguments than the number of formal arguments defined in the routine.

variable scope

Either private or public. Visibility of a private variable is restricted to the current compilation unit. A public variable may be referenced across multiple programs and modules.

variable-length string

A string of variable length. A variable-length string can be assigned a string of any length, up to the maximum length specified in the declaration. Pascal ignores any characters specified over the maximum.

`varying`

A string of variable length.

W

`wallclock`

A built-in function that returns the elapsed number of seconds since 00.00.00 GMT January 1, 1970.

`%warning`

A compiler directive that tells the compiler to print a string on the standard output as a warning. Programs that contain the `%warning` directive must be compiled with the `-xl` option.

X

`-xl` option

An option of the `pc` command that tells the compiler to implement Pascal as DOMAIN Pascal.

xor

A built-in function that returns the exclusive `or` of two integers.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X