## Contents

Preface. ................................................................. xiv

1. **Lexical Elements** ................................................. 1
   - Character Set ................................................. 1
   - Special Symbols ............................................. 2
   - Reserved Words ............................................. 3
   - Identifiers ................................................... 4
   - Comments .................................................... 6

2. **Data Types** ...................................................... 9
   - Summary of Data Format Differences .................... 10
   - Default Data Alignments and Padding .................. 10
   - Data Formats with \texttt{-calign} ......................... 11
   - Data Formats with \texttt{-xl} .............................. 12
   - Data Formats with \texttt{-calign} and \texttt{-xl} ........... 12
   - \texttt{real} ...................................................... 13
   - \texttt{real Variables} ...................................... 13
real Initialization ..................................... 13
real Constants .......................................... 14
Data Representation ..................................... 15
Integer ..................................................... 16
  Integer Variables ..................................... 17
  Integer Initialization ................................. 18
  Integer Constants ..................................... 18
  Data Representation .................................... 19
boolean .................................................... 20
  boolean Variables .................................... 20
  boolean Initialization ................................. 20
  boolean Constants .................................... 21
  Data Representation .................................... 21
Character .................................................. 22
  Character Variables .................................. 22
  Character Initialization ............................... 22
  Character Constants .................................. 23
  Data Representation .................................... 23
Enumerated Types ....................................... 23
  Enumerated Variables ................................. 24
  Data Representation .................................... 24
Subrange .................................................... 25
  Subrange Variables .................................... 25
  Data Representation .................................... 25
Record .............................................................. 26
  Record Variables .............................................. 27
  Record Initialization ......................................... 27
  Data Representation of Unpacked Records .............. 30
  Data Representation of Packed Records ............... 30
Array .............................................................. 34
  Array Variables ................................................. 34
  Array Initialization .......................................... 36
  Packed Arrays .................................................. 37
  Data Representation .......................................... 37
Set ................................................................. 38
  Set Variables .................................................... 38
  Set Initialization .............................................. 38
  Packed Sets ..................................................... 39
  Data Representation .......................................... 39
File ................................................................. 41
Pointer ............................................................ 41
  Standard Pointer .............................................. 41
  Universal Pointer ............................................. 42
  Procedure and Function Pointers ..................... 43
  Pointer Initialization ....................................... 45
  Data Representation .......................................... 45
3. Statements ..................................................... 47
  Standard Statements ......................................... 47
Statements Specific to Pascal

assert Statement ........................................ 48

case Statement ........................................ 51

exit Statement ........................................ 52

goto Statement ......................................... 54

next Statement ......................................... 56

otherwise Statement ................................. 58

return Statement ..................................... 59

with Statement ....................................... 60

4. Assignments and Operators .......................... 63

Data Type Assignments and Compatibility....... 63

String Assignments .................................. 64

Fixed- and Variable-Length Strings ............... 64

Null Strings .......................................... 65

String Constants ..................................... 65

Operators ................................................. 66

Arithmetic Operators ................................ 66

The mod Operator ................................... 66

Bit Operators ....................................... 68

boolean Operators .................................. 68

The and then Operator ............................... 69

The or else Operator ................................. 70

Set Operators ....................................... 71

Relational Operators ................................. 72
Relational Operators on Sets ............................... 72
The = and <> Operators on Records and Arrays ...... 73
String Operators ............................................. 75
Precedence of Operators .................................... 76

5. Program Declarations ................................. 77
Declarations ................................................. 77
Label Declaration ........................................... 77
Constant Declaration ....................................... 79
Type Declaration ............................................. 79
Variable Declaration ....................................... 80
Define Declaration ......................................... 83
Procedure and Function Headings .................... 84
Visibility ..................................................... 84
Parameter List .............................................. 85
Type Identifier ............................................. 89
Functions Returning Structured-Type Results .......... 89
Options ...................................................... 91

6. Built-In Procedures and Functions ................ 95
Standard Procedures and Functions ................... 95
Routines Specific to Pascal (Summary) ............... 96
Routines Specific to Pascal (Details) .................. 99
addr ......................................................... 99
append .................................................... 102
argc ....................................................... 105
argv................................................. 105
arshft .......................................... 107
asl................................................. 109
asr.................................................. 111
card............................................... 112
clock............................................... 113
close.............................................. 116
concat ........................................... 117
date............................................... 118
discard........................................... 120
expo............................................... 123
filesize.......................................... 124
firstof................................. 126
flush............................................. 130
getenv........................................... 132
getfile.......................................... 134
halt............................................... 136
in_range.................................... 138
index.......................................... 139
land............................................. 142
lastof........................................... 144
length......................................... 145
linelimit..................................... 147
lnot............................................. 149
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>lor</td>
<td>150</td>
</tr>
<tr>
<td>lshft</td>
<td>151</td>
</tr>
<tr>
<td>lsl</td>
<td>153</td>
</tr>
<tr>
<td>lsr</td>
<td>153</td>
</tr>
<tr>
<td>max</td>
<td>153</td>
</tr>
<tr>
<td>message</td>
<td>155</td>
</tr>
<tr>
<td>min</td>
<td>156</td>
</tr>
<tr>
<td>null</td>
<td>157</td>
</tr>
<tr>
<td>open</td>
<td>158</td>
</tr>
<tr>
<td>pcexit</td>
<td>161</td>
</tr>
<tr>
<td>random</td>
<td>162</td>
</tr>
<tr>
<td>read and readln</td>
<td>163</td>
</tr>
<tr>
<td>remove</td>
<td>166</td>
</tr>
<tr>
<td>reset</td>
<td>167</td>
</tr>
<tr>
<td>rewrite</td>
<td>168</td>
</tr>
<tr>
<td>rshft</td>
<td>171</td>
</tr>
<tr>
<td>seed</td>
<td>172</td>
</tr>
<tr>
<td>seek</td>
<td>174</td>
</tr>
<tr>
<td>sizeof</td>
<td>176</td>
</tr>
<tr>
<td>stlimit</td>
<td>180</td>
</tr>
<tr>
<td>stradd</td>
<td>182</td>
</tr>
<tr>
<td>substr</td>
<td>183</td>
</tr>
<tr>
<td>sysclock</td>
<td>184</td>
</tr>
<tr>
<td>tell</td>
<td>185</td>
</tr>
</tbody>
</table>
time ................................................. 187
trace .............................................. 189
trim .............................................. 191
Type Transfer .................................... 193
wallclock ....................................... 195
write and writeln .............................. 198
xor ............................................... 200

7. Input and Output ................................ 203
   Input and Output Routines ..................... 203
eof and eoln Functions .......................... 204
More About eoln ................................. 208
External Files and Pascal File Variables .... 210
   Permanent Files ............................... 210
   Temporary Files .............................. 211
input, output, and errout Variables ........... 211
   Properties of input, output, and errout Variables . 211
   Associating input with a File Other Than stdin .... 212
   Associating output with a File Other Than stdout . 212
   Associating errout with a File Other Than stderr . 212
Pascal I/O Library ............................... 213
Buffering of File Output ....................... 213
I/O Error Recovery ............................. 214
## Figures

| Figure 2-1 | 32-Bit Floating-Point Number | 15 |
| Figure 2-2 | 64-Bit Floating-Point Number | 15 |
| Figure 2-3 | 16-Bit Integer | 20 |
| Figure 2-4 | 32-Bit Integer | 20 |
| Figure 2-5 | true boolean Variable | 21 |
| Figure 2-6 | false boolean Variable | 22 |
| Figure 2-7 | 16-Bit Enumerated Variable | 24 |
| Figure 2-8 | Sample Enumerated Representation | 25 |
| Figure 2-9 | 16-Bit Subrange | 26 |
| Figure 2-10 | 32-Bit Subrange | 26 |
| Figure 2-11 | Sample Packed Record (Without \(-x\)) | 33 |
| Figure 2-12 | Small Set | 40 |
| Figure 2-13 | Large Set | 41 |
| Figure 2-14 | Pointer | 45 |
Tables

32-Bit Floating-Point Number 15
64-Bit Floating-Point Number 15
16-Bit Integer 20
32-Bit Integer 20
true boolean Variable 21
false boolean Variable 22
16-Bit Enumerated Variable 24
Sample Enumerated Representation 25
16-Bit Subrange 26
32-Bit Subrange 26
Sample Packed Record (Without –x1) 33
Small Set 40
Large Set 41
Pointer 45
Preface

The Pascal 4.0 compiler from SunSoft™ 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; they include:

- Separate compilation of programs and modules
- dbx (symbolic debugger) support, including the fix-and-continue functionality
- 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
• Specification of the direction of parameter passing as one of the following:
  ° Into a routine
  ° Out of a routine
  ° Both of the above

In addition, Pascal 4.0 contains a compiler switch, `-xl`, to provide compatibility with Apollo® DOMAIN® Pascal to ease the task of porting your Apollo Pascal applications to workstations.

Note – All references to Pascal in this manual refer to the Pascal 4.0 compiler unless otherwise indicated.

**Audience**

This manual provides reference material for the Pascal 4.0 compiler. To use this manual, you should be familiar with ISO standard Pascal and Solaris commands and concepts.

**Operating Environment**

For information on the operating environment, see the README file.

In a previous major release, this Pascal compiler also ran on Solaris™ 1.x. Some features remain in the documentation as being for Solaris 1.x only.

Note – For other release-specific information, refer to the README file that accompanies the product.

**Installation**

For instructions on how to install Pascal, refer to the documentation on installing the software, *Installing SunSoft Developer Products, SPARC/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 Declarations”
• Chapter 6, “Built-In Procedures and Functions”
• Chapter 7, “Input and Output”

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

This manual also has four appendixes:

• **Appendix A, “Overview of Pascal Extensions,”** summarizes the Pascal extensions to standard Pascal, and serves as a quick reference guide to the differences between Pascal and standard Pascal.

• **Appendix B, “Pascal and DOMAIN Pascal,”** lists the differences between Pascal and Apollo DOMAIN Pascal.

• **Appendix C, “Implementation Restrictions,”** describes Pascal features that are implementation-defined.

• **Appendix D, “Pascal Validation Summary Report,”** summarizes the features, errors, and extensions in the manufacturer’s statement of compliance for the validation of the Pascal Version 3.0.2 compiler.

A glossary and an index are included at the end of the manual.
Conventions Used in This Manual

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

Table P-1  BNF Meta Symbols

<table>
<thead>
<tr>
<th>Meta Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>::=</td>
<td>Defined as</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(a</td>
<td>b)</td>
</tr>
<tr>
<td>[ a ]</td>
<td>Zero or one instance of a</td>
</tr>
<tr>
<td>{ a }</td>
<td>Zero or more instances of a</td>
</tr>
<tr>
<td>'abc'</td>
<td>The characters abc</td>
</tr>
</tbody>
</table>

The following table describes the type styles and symbols used in this manual:

Table P-2  Typographic Conventions

<table>
<thead>
<tr>
<th>Typeface or Symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories; on-screen computer output</td>
<td>Edit your .login file. Use ls -a to list all files. hostname% You have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, contrasted with on-screen computer output</td>
<td>hostname% su Password:</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Command-line placeholder: replace with a real name or value</td>
<td>To delete a file, type rm filename.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new words or terms, or words to be emphasized</td>
<td>Read the User’s Guide. These are called class options. You must be root to do this.</td>
</tr>
</tbody>
</table>
Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-3  Shell Prompts

<table>
<thead>
<tr>
<th>Shell Prompt</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>C shell prompt</td>
<td>machine_name%</td>
</tr>
<tr>
<td>C shell superuser prompt</td>
<td>machine_name#</td>
</tr>
<tr>
<td>Bourne shell and Korn shell prompt</td>
<td>$</td>
</tr>
<tr>
<td>Bourne shell and Korn shell superuser prompt</td>
<td>#</td>
</tr>
</tbody>
</table>

Related Documentation

This manual is designed to accompany the following documents:

- The *Pascal 4.0 User’s Guide*, which describes how to use the compiler
- The *Pascal 4.0 Quick Reference*, which summarizes the compiler options

Both this manual and the *Pascal 4.0 User’s Guide* are available in the AnswerBook® system, an on-line documentation viewing tool that takes advantage of dynamically linked headings and cross-references. The *Installing SunSoft Developer Products (SPARC/Solaris)* manual shows you how to install AnswerBook.

Manual Page

Pascal 4.0 provides an on-line manual page (also known as a man page), on *pc*(1), that describes the Pascal compiler. This document is included in the Pascal package and must be installed with the rest of the software.

After you install the documentation, you can read about *pc* by entering the *man* command followed by the command name, as in:

```
hostname% man pc
```

README Files

The README default directory is: /opt/SUNWspro/README.s.
This directory contains the following files:

- A Pascal 4.0 README, called pascal, that describes the new features, software incompatibilities, and software bugs.

- A floating-point white paper, "What Every Scientist Should Know About Floating-Point Arithmetic," by David Goldberg, in PostScript™ format. The file is called floating-point.ps, and can be printed on any PostScript-compatible printer that has Palatino font. It can be viewed on-line by using the imagemtool command:

  hostname% imagemtool floating-point.ps

This paper is also available in the AnswerBook system.

### Other Related Documentation

Other reference material includes:

- Profiling Tools
- Numerical Computation Guide

### Documents in Hard Copy and in AnswerBook

The following table shows what documents are on-line, in hard copy, or both:

<table>
<thead>
<tr>
<th>Title</th>
<th>Hard Copy</th>
<th>On-Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal 4.0 User’s Guide</td>
<td>X</td>
<td>X (AnswerBook)</td>
</tr>
<tr>
<td>Pascal 4.0 Reference Manual</td>
<td>X</td>
<td>X (AnswerBook)</td>
</tr>
<tr>
<td>Pascal 4.0 Quick Reference</td>
<td>X</td>
<td>X (AnswerBook)</td>
</tr>
<tr>
<td>Installing SunSoft Developer Products (SPARC/Solaris)</td>
<td>X</td>
<td>X (AnswerBook)</td>
</tr>
<tr>
<td>Profiling Tools</td>
<td>X</td>
<td>(AnswerBook)</td>
</tr>
<tr>
<td>Numerical Computation Guide</td>
<td>X</td>
<td>(AnswerBook)</td>
</tr>
<tr>
<td>README</td>
<td>X</td>
<td>(CD-ROM)</td>
</tr>
<tr>
<td>What Every Scientist Should Know About Floating-Point Arithmetic</td>
<td>X</td>
<td>(AnswerBook and CD-ROM)</td>
</tr>
</tbody>
</table>

xix Pascal 4.0 Reference Manual
This chapter describes the symbols and words of a Pascal program. It contains the following sections:

<table>
<thead>
<tr>
<th>Character Set</th>
<th>page 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special Symbols</td>
<td>page 2</td>
</tr>
<tr>
<td>Reserved Words</td>
<td>page 3</td>
</tr>
<tr>
<td>Identifiers</td>
<td>page 4</td>
</tr>
<tr>
<td>Comments</td>
<td>page 6</td>
</tr>
</tbody>
</table>

**Character Set**

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

<table>
<thead>
<tr>
<th>Integer</th>
<th>INTEGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>INteger</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INTEger</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INTeger</td>
<td>INTEGER</td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
</tr>
</tbody>
</table>

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 in lowercase.

To map all keywords and identifiers to lowercase when you compile your program, use the following \texttt{pc} options:

\begin{itemize}
\item \texttt{-L} Maps all uppercase letters in keywords and identifiers to lowercase.
\item \texttt{-s} Performs the same action as \texttt{-L} and also produces warning diagnostics for nonstandard constructs and extensions.
\end{itemize}

See the \textit{Pascal 4.0 User’s Guide} for a complete description of \texttt{pc} and its options.

\section*{Special Symbols}

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

\begin{itemize}
\item \texttt{+ \ - \ * \ / \ = \ < \ > \ [ \ ] \ . \ , \ :=}
\item \texttt{: ; \ ( \ ) \ <> \ <= \ >= \ .. \ ^}
\end{itemize}

\begin{table}[ht]
\centering
\begin{tabular}{|c|p{5cm}|p{3cm}|}
\hline
Symbol & Description & Example \\
\hline
\texttt{~} & Bitwise not operator & \texttt{~4} \\
\hline
\texttt{&} & Bitwise and operator & \texttt{4 \& 3} \\
\hline
\texttt{|} & Bitwise or operator & \texttt{4 \| 3} \\
\hline
\texttt{!} & Bitwise or operator & \texttt{4 \! 3} \\
\hline
\texttt{#} & Specifies an integer value in a base other than base 10. & \texttt{p := 2#10111; \{ base 2 \}} \\
& & \texttt{f := 8#76543; \{ base 8 \}} \\
& Includes a file in the program. & \texttt{#include "globals.h"} \\
& & \texttt{#include "math_p.h"} \\
& Indicates a preprocessor command & \texttt{#ifdef DEBUGGING} \\
& & \texttt{writeln('Total :',i,sum); \#endif} \\
\hline
\texttt{%} & Indicates a \texttt{cppas} compiler directive & \texttt{%var one, two} \\
& & \texttt{%enable two} \\
\hline
\end{tabular}
\caption{Nonstandard Special Symbols}
\end{table}
Reserved Words

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

Table 1-2 Standard Reserved Words

<table>
<thead>
<tr>
<th>Pascal Standard Reserved Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
</tr>
<tr>
<td>array</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>case</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>div</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>downto</td>
</tr>
<tr>
<td>else</td>
</tr>
</tbody>
</table>

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.
Identifiers

In Pascal, you can include a dollar sign ($) and underscore (_) in an identifier name. The $ and _ can 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 1-3  Nonstandard Reserved Words

<table>
<thead>
<tr>
<th>Pascal Nonstandard Reserved Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>define</td>
</tr>
<tr>
<td>extern</td>
</tr>
<tr>
<td>external</td>
</tr>
<tr>
<td>module</td>
</tr>
<tr>
<td>otherwise</td>
</tr>
</tbody>
</table>

Table 1-4  Predeclared Standard Identifiers

<table>
<thead>
<tr>
<th>Pascal Predeclared Standard Identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
</tr>
<tr>
<td>arctan</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>chr</td>
</tr>
<tr>
<td>cos</td>
</tr>
<tr>
<td>dispose</td>
</tr>
<tr>
<td>eof</td>
</tr>
<tr>
<td>eoln</td>
</tr>
<tr>
<td>exp</td>
</tr>
</tbody>
</table>
You can 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
Mon Feb 20 15:13:17 1995  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
```

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 `*`).
You can nest comments in Pascal, that is, 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 cannot 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

This chapter describes the Pascal data types. Some data types represent different values when you compile your program with or without the -x1 option, and with or without the -calign option. The intent of the -x1 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.

This chapter contains the following sections:

<table>
<thead>
<tr>
<th>Summary of Data Format Differences</th>
<th>page 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>page 13</td>
</tr>
<tr>
<td>Integer</td>
<td>page 16</td>
</tr>
<tr>
<td>boolean</td>
<td>page 20</td>
</tr>
<tr>
<td>Character</td>
<td>page 22</td>
</tr>
<tr>
<td>Enumerated Types</td>
<td>page 23</td>
</tr>
<tr>
<td>Subrange</td>
<td>page 25</td>
</tr>
<tr>
<td>Record</td>
<td>page 26</td>
</tr>
<tr>
<td>Array</td>
<td>page 34</td>
</tr>
<tr>
<td>Set</td>
<td>page 38</td>
</tr>
<tr>
<td>File</td>
<td>page 41</td>
</tr>
<tr>
<td>Pointer</td>
<td>page 41</td>
</tr>
</tbody>
</table>
Summary of Data Format Differences

A few data formats, particularly of structured types, change when you use the Pascal compiler `-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.

Default Data Alignments and Padding

Here is a summary of the 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 when they are longer than 16 bits; otherwise, their alignment is two 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 on page 26 for examples.

Data Formats with -calign

With the -calign option, the data formats are:

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 sizes 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 four-byte sizes 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 following differences:

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:

- single, shortreal, double, and longreal data types
- real initialization in the variable declaration
- real constants without a digit after the decimal point

real Variables

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Maximum Value</th>
<th>Minimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>real (with -xl option)</td>
<td>32</td>
<td>3.402823e+38</td>
<td>1.401298e-45</td>
</tr>
<tr>
<td>real (without -xl option)</td>
<td>64</td>
<td>1.79769313486231470e+308</td>
<td>4.94065645841246544e-324</td>
</tr>
<tr>
<td>single</td>
<td>32</td>
<td>3.402823e+38</td>
<td>1.401298e-45</td>
</tr>
<tr>
<td>shortreal</td>
<td>32</td>
<td>3.402823e+38</td>
<td>1.401298e-45</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>1.79769313486231470e+308</td>
<td>4.94065645841246544e-324</td>
</tr>
<tr>
<td>longreal</td>
<td>64</td>
<td>1.79769313486231470e+308</td>
<td>4.94065645841246544e-324</td>
</tr>
</tbody>
</table>

This example declares five real variables:

```pascal
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:
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:

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

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

The example in the following section 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 constant:

```
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; Figure 2-2 shows the representation of a 64-bit floating point number.

Figure 2-1 32-Bit Floating-Point Number

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent + 127</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-2 64-Bit Floating-Point Number

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent + 1023</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32</td>
<td></td>
</tr>
</tbody>
</table>

A real number is represented by this form:

\[ (-1)^{\text{sign}} \times 2^{\text{exponent bias}} \times 1.f \]

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

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero (signed)</td>
<td>Represented by an exponent of zero and a fraction of zero.</td>
</tr>
<tr>
<td>Subnormal number</td>
<td>Represented by ((-1)^{\text{sign}} \times 2^{\text{1-bias}} \times 0.f), where (f) is the bits in the significand.</td>
</tr>
<tr>
<td>Not a Number (NaN)</td>
<td>Represented by the largest value that the exponent an assume (all ones), and a nonzero fraction.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>32-bit Floating-Point Number</th>
<th>64-bit Floating-Point Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>00000000</td>
<td>0000000000000000</td>
</tr>
<tr>
<td>-0</td>
<td>80000000</td>
<td>8000000000000000</td>
</tr>
<tr>
<td>+1.0</td>
<td>3F800000</td>
<td>3FF0000000000000</td>
</tr>
<tr>
<td>-1.0</td>
<td>BF800000</td>
<td>BFF0000000000000</td>
</tr>
<tr>
<td>+2.0</td>
<td>40000000</td>
<td>4000000000000000</td>
</tr>
<tr>
<td>+3.0</td>
<td>40400000</td>
<td>4080000000000000</td>
</tr>
<tr>
<td>+Infinity</td>
<td>7F800000</td>
<td>7FF0000000000000</td>
</tr>
<tr>
<td>-Infinity</td>
<td>FF800000</td>
<td>FFF0000000000000</td>
</tr>
<tr>
<td>NaN</td>
<td>7Fxxxxxx</td>
<td>7Fxxxxxxxxxxxxxx</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Type</th>
<th>Number of Bits</th>
<th>Maximum Value</th>
<th>Minimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer (without -xl option)</td>
<td>32</td>
<td>2,147,483,647</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>integer (with -xl option)</td>
<td>16</td>
<td>32,767</td>
<td>-32,768</td>
</tr>
<tr>
<td>integer16</td>
<td>16</td>
<td>32,767</td>
<td>-32,768</td>
</tr>
<tr>
<td>integer32</td>
<td>32</td>
<td>2,147,483,647</td>
<td>-2,147,483,648</td>
</tr>
</tbody>
</table>

This example declares three integer variables:

```pascal
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:

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

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

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

You can 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:

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

Integer Constants

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

Here is an example:

```
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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Without −x1</th>
<th>With −x1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bits</td>
<td>Value</td>
</tr>
<tr>
<td>maxint</td>
<td>32</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>minint</td>
<td>32</td>
<td>-2,147,483,648</td>
</tr>
</tbody>
</table>

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 can optionally put a positive sign (+) or negative sign (−) before *base*. The sign applies to the entire number, not the base.

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

\[
\begin{align*}
\text{power} & := 2\#10111; \quad (* \text{binary (base 2)} *) \\
\text{fraction_of_c} & := -8\#76543; \quad (* \text{octal (base 8)} *) \\
\text{percentage} & := +16\#fd9c; \quad (* \text{hexadecimal (base 16)} *) 
\end{align*}
\]

Data Representation

Pascal represents integer, integer16, and integer32 data types in two's 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.
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:

```pascal
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:

```pascal
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:

```pascal
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:

```pascal
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; Figure 2-6 shows how Pascal represents a false boolean variable.

```
0 0 0 0 0 0 1 1
```

*Figure 2-5  true boolean Variable*
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:

```pascal
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:

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

This example initializes the variable `pass` to A and `fail` to 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`:

```pascal
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**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>minchar</td>
<td>Equal to <code>char(0)</code></td>
</tr>
<tr>
<td>maxchar</td>
<td>Equal to <code>char(255)</code></td>
</tr>
<tr>
<td>bell</td>
<td>Equal to <code>char(7)</code> (which makes your terminal beep)</td>
</tr>
<tr>
<td>tab</td>
<td>Equal to <code>char(9)</code> (which makes a tab character)</td>
</tr>
</tbody>
</table>

**Data Representation**

Pascal allocates one byte for each character variable.

**Enumerated Types**

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.

```
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

  Byte 0  Byte 1
  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
```

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.
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 “Integer Variables” on page 17 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.

---

**Figure 2-8**  Sample Enumerated Representation

<table>
<thead>
<tr>
<th>Red</th>
<th>Green</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Blue</th>
<th>Orange</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 1 0</td>
<td>0 0 0 0 0 0 1 1</td>
</tr>
</tbody>
</table>
Table 2-7  Subrange Data Representation

<table>
<thead>
<tr>
<th>Minimum/Maximum Range</th>
<th>Without $\times 1$ (Bits)</th>
<th>With $\times 1$ (Bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..127</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>-128..127</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>0..255</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>-32768..32767</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>0..65536</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>-2,147,483,648..2,147,483,647</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

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

```pascal
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.
  
  ```pascal
  [ 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.
  
  ```pascal
  [ 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`.

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

```pascal
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: ( il6: integer16;
        d: double )
    end;
var
{ Initialization by name. }
  rec1: record_type :=
    [st := ['a', 'b', 'c'],
     c := 'A',
     z := 'ARRAY1',
     colors := green,
     il6 := 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 }
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 – In packed records, bit-aligned fields do not cross word boundaries.
### Packed Record Storage Without the \(-x1\) Option

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>4 bytes</td>
<td>4 bytes</td>
</tr>
<tr>
<td>integer16</td>
<td>2 bytes</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>integer32</td>
<td>4 bytes</td>
<td>4 bytes</td>
</tr>
<tr>
<td>real</td>
<td>8 bytes</td>
<td>8 bytes</td>
</tr>
<tr>
<td>single</td>
<td>4 bytes</td>
<td>4 bytes</td>
</tr>
<tr>
<td>shortreal</td>
<td>4 bytes</td>
<td>4 bytes</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>8 bytes</td>
</tr>
<tr>
<td>longreal</td>
<td>8 bytes</td>
<td>8 bytes</td>
</tr>
<tr>
<td>boolean</td>
<td>1 bit</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>char</td>
<td>1 byte</td>
<td>1 byte</td>
</tr>
<tr>
<td>enumerated</td>
<td>Number of bits required to represent the highest ordinal value</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>subrange of char</td>
<td>1 byte</td>
<td>1 byte</td>
</tr>
<tr>
<td>all other subrange</td>
<td>Number of bits required to represent the highest ordinal value</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>set of cardinality (\leq 32)</td>
<td>One bit per element</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>set of cardinality (&gt; 32)</td>
<td>Same as if unpacked</td>
<td>4 bytes</td>
</tr>
<tr>
<td>array</td>
<td>Requires the same space required by the base type of the array</td>
<td>Same as element type</td>
</tr>
</tbody>
</table>


### Packed Record Storage with the –xl Option

**Table 2-9** Packed Record Storage with –xl

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>4 bytes</td>
<td>4 bytes</td>
</tr>
<tr>
<td>integer</td>
<td>2 bytes</td>
<td>Bit-aligned</td>
</tr>
</tbody>
</table>

### Packed Record Storage with the –calign Option

**Table 2-10** Packed Record Storage with –calign

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer16</td>
<td>2 bytes</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

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.

```pascal

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Size (Bits)</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>16</td>
<td>16 bit-aligned</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>sm</td>
<td>8</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>med</td>
<td>16</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>lg (without (-xl))</td>
<td>32</td>
<td>32 bit-aligned</td>
</tr>
<tr>
<td>lg (with (-xl))</td>
<td>16</td>
<td>16 bit-aligned</td>
</tr>
<tr>
<td>se</td>
<td>4</td>
<td>Bit-aligned</td>
</tr>
<tr>
<td>x</td>
<td>32</td>
<td>32 bit-aligned</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alfa</td>
<td>An array of char 10 characters long.</td>
</tr>
<tr>
<td>string</td>
<td>An array of char 80 characters long.</td>
</tr>
<tr>
<td>varying</td>
<td>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 65,535</td>
</tr>
</tbody>
</table>

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.
The following program demonstrates the differences between the fixed-length and varying data types:

```pascal
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.
```
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 the following four different formats:

- Supply the lower and upper bounds in the initialization.

```
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 can use this format only when you also supply the initial values.

```
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.

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

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.

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

This code initializes all the first 50 values of `int2` to `1` and the second 50 values to `2`. 
• Use the repeat count feature \( * \) of constant to initialize all remaining array elements to the value of constant.

```pascal
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],
    ];
```

This example initializes all 100 elements of \texttt{int4} to 327. The example also initializes the multidimensional array \texttt{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.

When you initialize an array in the \texttt{var} declaration, the compiler sets those elements for which it doesn’t find data to zero.

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

**Packed Arrays**

Although you can define an array as \texttt{packed}, it has 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 \texttt{-calign} option, the size of all arrays is the size of each element times the number of elements. When you use the \texttt{-calign} and \texttt{-xl} options together, arrays are the same as with \texttt{-calign} alone, except the size of an array of \texttt{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; you can then 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:

```pascal
type
    character_set = set of char;
var
    letters: character_set;
```

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:

```pascal
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:

```
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 has 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 32,768.

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

<table>
<thead>
<tr>
<th>Set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set of 0..15</td>
<td>This set requires 16 bits because 15 is the maximal element, and 15 + 1 = 16.</td>
</tr>
<tr>
<td>set of 0..16</td>
<td>This set requires 32 bits because 16 is the maximal element. 16 + 1 = 17, and the next multiple of 16 above 17 is 32.</td>
</tr>
<tr>
<td>set of 14..15</td>
<td>This set requires 16 bits because 15 is the element, and 15 + 1 = 16.</td>
</tr>
<tr>
<td>set of char</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

```pascal
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.

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

**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, to assign a pointer of one type to another, or to weaken type checking when passing parameters of pointer types.

When the type of a formal parameter is `univ_ptr`, the type of the corresponding actual parameter can be of any pointer type, or vice versa.

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

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

```pascal
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
```

**Procedure and Function Pointers**

The following is an example that shows how to use procedure and function pointers in Pascal.
The Pascal program, `pointer.p`, which demonstrates how to print out enumerated values using procedure pointers.

```pascal
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 }
```
Pointer Initialization

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

```pascal
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](image)
This chapter describes Pascal statements in the following sections:

<table>
<thead>
<tr>
<th>Standard Statements</th>
<th>page 47</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statements Specific to Pascal</td>
<td>page 47</td>
</tr>
</tbody>
</table>

**Standard Statements**

Pascal supports all standard statements. Pascal also supports extensions to:

- `assert` next
- `case` otherwise
- `exit` return
- `goto` with

**Statements Specific to Pascal**

Table 3-1 summarizes the nonstandard Pascal statements and standard statements with nonstandard features. Detailed descriptions and examples of each statement follow.
assert Statement

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

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 uses an `assert` statement to test whether `num` is greater than 0 and less than or equal to `MAX_STUDENTS`:

```plaintext
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:

```plaintext
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
else begin
    writeln('Error message. ');
    halt
end
```
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.

```pascal
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
```
case Statement

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

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 can 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.
The Pascal program, otherwise.p, which 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. The program specifies all digits between 0 and 9 as the range '0'..'9'.

```pascal
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.
```

**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.

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

You receive a compile-time error if you use this statement anywhere but in a for, while, or repeat loop.
The Pascal program, exit.p

```pascal
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% cc exit.p
hostname% a.out
Enter a number:  13
Number WAS found:          13
```
**goto Statement**

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

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

```pascal
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;
      end;
      sum := sum + points;
    end;
  end;
  writeln('Average GPA = ', sum / (num - 1));
end.
```

Identifiers as Targets (Screen 1 of 2)
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.

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 receive a compile-time error if you use this statement anywhere but in a `for`, `while`, or `repeat` loop.
The Pascal program, `next.p`, which also uses the `otherwise` statement.

```pascal
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)
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 in “**case Statement**” on page 51 for additional information.
return Statement

The return statement prematurely ends a procedure or a function.

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.

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
```

### with Statement

Pascal supports the standard `with` statement plus an alternative format.

The following is an example that illustrates how to use a `with` statement in Pascal.
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 ');
  writeln(new_patient.FirstName: 10);
  writeln(new_patient.LastName: 10, '.');
  write('The old patient is ');
  writeln(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.
```
Assignments and Operators

This chapter describes the different types of assignments and operators in Pascal. It contains the following sections:

<table>
<thead>
<tr>
<th>Data Type Assignments and Compatibility</th>
<th>page 63</th>
</tr>
</thead>
<tbody>
<tr>
<td>String Assignments</td>
<td>page 64</td>
</tr>
<tr>
<td>Operators</td>
<td>page 66</td>
</tr>
<tr>
<td>Precedence of Operators</td>
<td>page 76</td>
</tr>
</tbody>
</table>

Data Type Assignments and Compatibility

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

<table>
<thead>
<tr>
<th>Type of Variable/Parameter</th>
<th>Type of Assignment-Compatible Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>real, single, shortreal</td>
<td>real, single, shortreal, double, longreal, any integer type†</td>
</tr>
<tr>
<td>double, longreal</td>
<td>real, single, shortreal, double, longreal, any integer type</td>
</tr>
<tr>
<td>integer, integer16, integer32</td>
<td>integer, integer16, integer32</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>enumerated</td>
<td>Same enumerated type</td>
</tr>
<tr>
<td>subrange</td>
<td>Base type of the subrange</td>
</tr>
<tr>
<td>record</td>
<td>Record of the same type</td>
</tr>
<tr>
<td>array</td>
<td>Array with the same type</td>
</tr>
<tr>
<td>set</td>
<td>Set with compatible base type</td>
</tr>
<tr>
<td>pointer</td>
<td>Pointer to an identical type, univ_ptr</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Assignment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varying := '';</td>
<td>The compiler assigns the null string to the variable-length string. The length of the variable-length string equals zero.</td>
</tr>
<tr>
<td>array of char := '';</td>
<td>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.</td>
</tr>
<tr>
<td>char := '';</td>
<td>It is illegal to assign a null string to a char variable. Use chr(0) instead.</td>
</tr>
<tr>
<td>String concatenation</td>
<td>In a string concatenation expression such as:</td>
</tr>
<tr>
<td></td>
<td>S := 'hello' + '' + S;</td>
</tr>
<tr>
<td></td>
<td>'' is treated as the additive identity (as nothing).</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operands</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>integer or real</td>
<td>integer or real</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>integer or real</td>
<td>integer or real</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>integer or real</td>
<td>integer or real</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>integer or real</td>
<td>real</td>
</tr>
<tr>
<td>div</td>
<td>truncated division</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>mod</td>
<td>modulo</td>
<td>integer</td>
<td>integer</td>
</tr>
</tbody>
</table>

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:

$-1 \times \text{remainder of } |i| \text{ divided by } |j|$
The Pascal program, `mod.p`, which computes $i \mod j$

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operands</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>~</code></td>
<td>bitwise not</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>bitwise and</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>bitwise or</td>
<td>integer</td>
</tr>
<tr>
<td><code>!</code></td>
<td>bitwise or (same as `</td>
<td>`)</td>
<td>integer</td>
</tr>
</tbody>
</table>

**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 `!` 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**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operands</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>~</code></td>
<td>bitwise not</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>bitwise and</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>`</td>
<td>`</td>
<td>bitwise or</td>
<td>integer</td>
</tr>
<tr>
<td><code>!</code></td>
<td>bitwise or (same as `</td>
<td>`)</td>
<td>integer</td>
</tr>
</tbody>
</table>

**boolean Operators**

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operands</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>Conjunction</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>and then</td>
<td>Similar to boolean and</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>not</td>
<td>Negation</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>or</td>
<td>Disjunction</td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>or else</td>
<td>Similar to boolean or</td>
<td>boolean</td>
<td>boolean</td>
</tr>
</tbody>
</table>

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 \( \text{odd}(y) \) before it evaluates \( \text{odd}(x) \):

\[
\text{odd}(x) \text{ and } \text{odd}(y)
\]

However, when you use the following syntax, the compiler always evaluates \( \text{odd}(x) \) first:

\[
\text{odd}(x) \text{ and then } \text{odd}(y)
\]

If \( \text{odd}(x) \) is false, \( \text{odd}(y) \) is not evaluated.

Note – You cannot insert comments between the and and the then operators.
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 – You cannot insert comments between the or and the else 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**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operands</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Set union</td>
<td>Any set type</td>
<td>Same as operands</td>
</tr>
<tr>
<td>-</td>
<td>Set difference</td>
<td>Any set type</td>
<td>Same as operands</td>
</tr>
<tr>
<td>*</td>
<td>Set intersection</td>
<td>Any set type</td>
<td>Same as operands</td>
</tr>
<tr>
<td>in</td>
<td>Member of a specified set 2nd arg; any set type 1st arg: base type of 2nd arg</td>
<td>boolean</td>
<td></td>
</tr>
</tbody>
</table>
Relational Operators

The relational operators are given in Table 4-8. 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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operand</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
<td>Any real, integer, boolean, char, record, array, set, or pointer type</td>
<td>boolean</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
<td>Any real, integer, boolean, char, record, array, set, or pointer type</td>
<td>boolean</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>Any real, integer, boolean, char, string, or set type</td>
<td>boolean</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</td>
<td>Any real, integer, boolean, char, string, or set type</td>
<td>boolean</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>Any real, integer, boolean, char, string, or set type</td>
<td>boolean</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
<td>Any real, integer, boolean, char, string, or set type</td>
<td>boolean</td>
</tr>
</tbody>
</table>

Relational Operators on Sets

Use the relational operators to compare sets of identical types. The result is a boolean (true or false) value.
The = and <> Operators on Records and Arrays

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

- 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, 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 if a set 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
```
The Pascal program, `compare.p`, which makes comparisons among records

```pascal
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;
  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)
Comparing Records (Screen 2 of 2)

```pascal
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 }
```

The commands to compile and execute `compare.p`

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

**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.
Precedence of Operators

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

<table>
<thead>
<tr>
<th>Operators</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>~, not,</td>
<td>Highest</td>
</tr>
<tr>
<td>*, /, div, mod, and, &amp;</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>, !, +, -, or,</td>
</tr>
<tr>
<td>=, &lt;&gt;, &lt;, &lt;=, &gt;, &gt;=, in,</td>
<td>.</td>
</tr>
<tr>
<td>or else, and then</td>
<td>Lowest</td>
</tr>
</tbody>
</table>
Program Declarations

This chapter describes Pascal program declarations. It contains the following sections:

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarations</td>
<td>77</td>
</tr>
<tr>
<td>Procedure and Function Headings</td>
<td>84</td>
</tr>
</tbody>
</table>

Declarations

This section describes the label, constant, type, variable, and define declarations. Procedure and function declarations are described in Chapter 6, “Built-In Procedures and Functions.”

Label Declaration

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

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

```pascal
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 }
```

Pascal 4.0 Reference Manual
Constant Declaration

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

The value of expression can be a compile-time evaluable expression. It can 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.

```pascal
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.
Unlike standard Pascal, in Pascal, you can 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.

In the variable declaration, you can specify the variable scope, attributes, and initial values. In most cases, you do 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.

You can also use the `define/extern` declaration to declare a variable as `public`, and the `static` attribute to declare a variable as `private`. See Appendix A, “Overview of Pascal Extensions,” for information on `define/extern`. 
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`.

<table>
<thead>
<tr>
<th>public var</th>
<th>total: single := 100.00;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>quantity: integer16 := 25;</td>
</tr>
<tr>
<td>private var</td>
<td>score: integer16 := 99;</td>
</tr>
</tbody>
</table>

**Attributes**

The variable attributes determine how to allocate the variable and its scope. They include `static`, `extern`, and `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 can only initialize a local variable, that is, a variable declared in a procedure, in the `var` declaration if you also declare it as `static`.

This code declares both `public` and `private` variables.
The Pascal program, static.p

```pascal
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
```

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 cannot initialize `extern` variables. See the *Pascal 4.0 User’s Guide*, which describes separately compiled programs and modules; it also contains examples of the `extern` attribute.
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 4.0 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`.

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

```pascal
var
  x: array[1..5, 1..3] of real := [[* of 0.0],[* of 0.0]];
  year, zeta: integer := 0;
  sunny: boolean := false;
  cl: 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`.

```pascal
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.
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 can 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 4.0 User’s Guide for examples of the define declaration.

Procedure and Function Headings

This section discusses the visibility, parameters, the type identifier, functions, and options for procedure and function headings.

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 can reference that routine in another program or module unit. Declaring a routine as private restricts its accessibility to the current compilation unit.

You can 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 Appendix A, “Overview of Pascal Extensions.”
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 4.0 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.

Parameters: in, out, and in out

The in, out, and in out parameters are extensions to the standard, which are used to specify the direction of parameter passing:
<table>
<thead>
<tr>
<th>Parameter Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>in</code></td>
<td>Indicates that the parameter can only pass a value into the routine. The parameter is, in effect, a read-only variable. You cannot assign a value to an <code>in</code> parameter, nor can you pass an <code>in</code> parameter as an argument to another procedure that expects a <code>var</code>, <code>out</code>, or <code>in out</code> argument.</td>
</tr>
<tr>
<td><code>out</code></td>
<td>Indicates that the parameter is used to pass values out of the routine. In effect, declaring a parameter as <code>out</code> informs the compiler that the parameter has no initial value, and that assignments to the parameter are retained by the caller.</td>
</tr>
<tr>
<td><code>in out</code></td>
<td>Indicates that the parameter can both take in values and pass them back out. An <code>in out</code> parameter is equivalent to a <code>var</code> parameter.</td>
</tr>
</tbody>
</table>
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.

```pascal
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 }
```
var Parameters

With standard conformance options (-s, -V0, -V1), var parameters are the same in standard Pascal and Pascal. By default, the Apollo-like var compatibility approach applies: actual and formal records and arrays should be of the same type; other types of var must be of the same length.

For example, all pointer types are compatible to univ_ptr, and vice versa. See “Universal Pointer” on page 42. Subranges -128...127 and 0...127 are also var-compatible.

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:

```pascal
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.

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

```pascal
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 can only represent the simple types of value, ordinal, or real.

**Functions Returning Structured-Type Results**

If a Pascal function returns the result of a structured type, for example, an array, a record, a string, or some combination of these, you can construct or update the result, component-by-component, using assignments of the form:

```
F S1 ... SN := E
```

where:

- F is the function name
- S1, ..., SN are appropriate component selectors
- E is the result component value
Standard Pascal allows assignments to the whole function result variable only, that is, \( F := E \), which may not be feasible or efficient enough, since you may have to declare and initialize extra structured-type variables.

**Example 1: A Function That Returns 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 following example, 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.
Example 2: A Function that Returns Arrays of Records
(Complex Vector Addition)

```pascal
program complex_vectors;

type
complex = record re, im: real end;
compl_vect = array [1..10] of complex;

function add (var a, b: compl_vect): compl_vect;
var i: integer;
begin
  for i:= 1 to 10 do
    begin
      add[i].re:= a[i].re + b[i].re;
      add[i].im:= a[i].im + b[i].im;
    end;
end; { add }

var V1, V2, V3: compl_vect;
begin
  ...
  V1:= add (V2, V3);
  ...
end. { complex_vectors }
```

Options

Pascal supplies the standard `forward` routine option and the nonstandard options, `extern`, `external`, `internal`, `variable`, and `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 4.0 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

Using the variable option, you can 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 cannot 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 }
```
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.

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.
```
This chapter describes the built-in procedures and functions Pascal supports. It starts with two major sections:

<table>
<thead>
<tr>
<th>Standard Procedures and Functions</th>
<th>page 95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Routines Specific to Pascal (Summary)</td>
<td>page 96</td>
</tr>
<tr>
<td>Routines Specific to Pascal (Details)</td>
<td>page 99</td>
</tr>
</tbody>
</table>

A third section, beginning on page 99, lists the nonstandard routines alphabetically and contains detailed descriptions and examples of each routine.

**Standard Procedures and Functions**

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

**Table 6-1 Standard Procedures**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>dispose</td>
<td>page</td>
</tr>
<tr>
<td>get</td>
<td>put</td>
</tr>
<tr>
<td>new</td>
<td>read</td>
</tr>
<tr>
<td>pack</td>
<td></td>
</tr>
<tr>
<td>readln</td>
<td></td>
</tr>
<tr>
<td>resey</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td></td>
</tr>
<tr>
<td>writeln</td>
<td></td>
</tr>
</tbody>
</table>
Routines Specific to Pascal (Summary)

This section 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 through Table 6-8 summarize these Pascal routines.

**Table 6-2 Standard Functions**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td></td>
</tr>
<tr>
<td>arctan</td>
<td></td>
</tr>
<tr>
<td>chr</td>
<td></td>
</tr>
<tr>
<td>cos</td>
<td></td>
</tr>
<tr>
<td>eof</td>
<td></td>
</tr>
<tr>
<td>eoln</td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td></td>
</tr>
<tr>
<td>ln</td>
<td></td>
</tr>
<tr>
<td>odd</td>
<td></td>
</tr>
<tr>
<td>ord</td>
<td></td>
</tr>
<tr>
<td>pred</td>
<td></td>
</tr>
<tr>
<td>succ</td>
<td></td>
</tr>
<tr>
<td>round</td>
<td></td>
</tr>
<tr>
<td>sin</td>
<td></td>
</tr>
<tr>
<td>sqr</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td></td>
</tr>
<tr>
<td>trunc</td>
<td></td>
</tr>
</tbody>
</table>

**Table 6-3 Nonstandard Arithmetic Routines**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>Returns the address of a variable, constant, function, or procedure.</td>
</tr>
<tr>
<td>card</td>
<td>Returns the cardinality of a set.</td>
</tr>
<tr>
<td>expo</td>
<td>Calculates the exponent of a variable.</td>
</tr>
<tr>
<td>firstof</td>
<td>Returns the first possible value of a type or variable.</td>
</tr>
<tr>
<td>in_range</td>
<td>Determines whether a value is in the defined integer subrange.</td>
</tr>
<tr>
<td>lastof</td>
<td>Returns the last possible value of a type or variable.</td>
</tr>
<tr>
<td>max</td>
<td>Returns the larger of two expressions.</td>
</tr>
<tr>
<td>min</td>
<td>Returns the smaller of two expressions.</td>
</tr>
<tr>
<td>random</td>
<td>Generates a random number between 0.0 and 1.0.</td>
</tr>
<tr>
<td>seed</td>
<td>Resets the random number generator.</td>
</tr>
<tr>
<td>sizeof</td>
<td>Returns the size of a designated type or variable.</td>
</tr>
</tbody>
</table>
Table 6-4  Nonstandard Bit Shift Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arshft</td>
<td>Does an arithmetic right shift of an integer.</td>
</tr>
<tr>
<td>asl</td>
<td>Does an arithmetic left shift of an integer.</td>
</tr>
<tr>
<td>asr</td>
<td>Identical to arshft.</td>
</tr>
<tr>
<td>land</td>
<td>Returns the bitwise \texttt{and} of two integers.</td>
</tr>
<tr>
<td>lnot</td>
<td>Returns the bitwise \texttt{not} of an integer.</td>
</tr>
<tr>
<td>lor</td>
<td>Returns the inclusive \texttt{or} of two integers.</td>
</tr>
<tr>
<td>lshft</td>
<td>Does a logical left shift of an integer.</td>
</tr>
<tr>
<td>lsl</td>
<td>Identical to lshft.</td>
</tr>
<tr>
<td>lsr</td>
<td>Identical to rshft.</td>
</tr>
<tr>
<td>rshft</td>
<td>Does a logical right shift of an integer.</td>
</tr>
<tr>
<td>xor</td>
<td>Returns the exclusive \texttt{or} of two integers.</td>
</tr>
</tbody>
</table>

Table 6-5  Nonstandard Character String Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>concat</td>
<td>Concatenates two strings.</td>
</tr>
<tr>
<td>index</td>
<td>Returns the position of the first occurrence of a string or character inside another string.</td>
</tr>
<tr>
<td>length</td>
<td>Returns the length of a string.</td>
</tr>
<tr>
<td>stradd</td>
<td>Adds a string to another string.</td>
</tr>
<tr>
<td>substr</td>
<td>Extracts a substring from a string.</td>
</tr>
<tr>
<td>trim</td>
<td>Removes all trailing blanks in a character string.</td>
</tr>
</tbody>
</table>
Table 6-6  Nonstandard Input and Output Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>Opens a file for modification at its end.</td>
</tr>
<tr>
<td>close</td>
<td>Closes a file.</td>
</tr>
<tr>
<td>filesize</td>
<td>Returns the current size of a file.</td>
</tr>
<tr>
<td>flush</td>
<td>Writes the output buffered for a Pascal file into the associated operating system file.</td>
</tr>
<tr>
<td>getfile</td>
<td>Returns a pointer to the C standard I/O descriptor associated with a Pascal file.</td>
</tr>
<tr>
<td>linelimit</td>
<td>Terminates program execution after a specified number of lines has been written into a text file.</td>
</tr>
<tr>
<td>message</td>
<td>Writes the specified information on stderr.</td>
</tr>
<tr>
<td>open</td>
<td>Associates an external file with a file variable.</td>
</tr>
<tr>
<td>remove</td>
<td>Removes the specified file.</td>
</tr>
<tr>
<td>seek</td>
<td>Performs random access to a file, changing its current position.</td>
</tr>
<tr>
<td>tell</td>
<td>Returns the current position of a file.</td>
</tr>
</tbody>
</table>

Table 6-7  Extensions to Standard Input and Output Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read and</td>
<td>Reads in boolean variables, fixed- and variable-length strings, and enumerated types from the standard input.</td>
</tr>
<tr>
<td>readline</td>
<td></td>
</tr>
<tr>
<td>reset and</td>
<td>Accepts an optional second argument, an operating system file name.</td>
</tr>
<tr>
<td>rewrite</td>
<td></td>
</tr>
<tr>
<td>write and</td>
<td>Outputs enumerated type values to the standard output.</td>
</tr>
<tr>
<td>writeln</td>
<td>Outputs expressions in octal or hexadecimal.</td>
</tr>
<tr>
<td></td>
<td>Allows negative field widths.</td>
</tr>
</tbody>
</table>
Table 6-8  Miscellaneous Nonstandard Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>Returns the number of arguments passed to the program.</td>
</tr>
<tr>
<td>argv</td>
<td>Assigns the specified program arguments a string variable.</td>
</tr>
<tr>
<td>clock</td>
<td>Returns the user time consumed by this process.</td>
</tr>
<tr>
<td>date</td>
<td>Fetches the current date.</td>
</tr>
<tr>
<td>discard</td>
<td>Explicitly discards the return value of a function.</td>
</tr>
<tr>
<td>getenv</td>
<td>Returns the value associated with an environment name.</td>
</tr>
<tr>
<td>halt</td>
<td>Terminates program execution.</td>
</tr>
<tr>
<td>null</td>
<td>Performs no operation.</td>
</tr>
<tr>
<td>pcexit</td>
<td>Terminates the program and returns an exit code.</td>
</tr>
<tr>
<td>stlimit</td>
<td>Terminates program execution if a specified number of statements have been executed in the current loop.</td>
</tr>
<tr>
<td>sysclock</td>
<td>Returns the system time consumed by this process.</td>
</tr>
<tr>
<td>time</td>
<td>Retrieves the current time.</td>
</tr>
<tr>
<td>trace</td>
<td>Prints a stack traceback.</td>
</tr>
<tr>
<td>Type transfer</td>
<td>Changes the data type of a variable or expression.</td>
</tr>
<tr>
<td>wallclock</td>
<td>Returns the elapsed number of seconds since 00:00:00 GMT January 1, 1970.</td>
</tr>
</tbody>
</table>

**Routines Specific to Pascal (Details)**

Described in this section are the detailed descriptions for each of the Pascal-specific routines: its syntax, arguments, and return value. Comments and an example are also included.

**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

In Pascal, you can apply `addr` to a variable, function, or procedure with dynamic extent such as local variables and nested functions or procedures. 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 results 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, do not use the `-H` option. The `-H` option makes sure that all pointers used point into the heap.
Example

The Pascal program, addr.p

```pascal
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.
```

append

The append function allows a file to be modified, and sets the current position to the end of the file.

**Syntax**

```pascal
append(file, filename)
```

**Arguments**

- `file` is a variable with the `text` or `file` data type.
- `filename`, which is optional, is a string of fixed or variable length, or a string constant.

**Return Value**

append does not return any values.

**Comments**

For example, this code associates the Pascal file data with the operating system file, `existent`:

```pascal
append(data, 'existent');
```

If you do not pass an optional second argument, Pascal creates a new temporary file, which is deleted when the program is terminated.

See also the sections: “reset,” “rewrite,” and “close.”
Example

The example that follows shows how to use `append`.
The Pascal program, files.p

```pascal
program files_example(input, output);
const
  MaxLength = 80;
var
  f: text;
  line: varying [MaxLength] of char;
begin
  rewrite(f, 'poem.txt');
  writeln('Enter a lines of text and hit Control+D to end the job.');
  while not eof do begin
    readln(line);
    writeln(f, line);
  end;
  close(f);
  writeln;
write ln('There 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);
reset(input); { Because Control+D close input }
append(f, 'poem.txt');
writeln('Append a lines of text and hit Control+D to end the job.');
while not eof do begin
  readln(line);
  writeln(f, line);
end;
close(f);
writeln;
writeln('There are the lines of all text you input:');
reset(f, 'poem.txt');
while not eof(f) do begin
  readln(f, line);
  writeln(line);
end;
close(f);
end.
```
**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 on page 105.

**Example**

See the example in the `argv` listing page 105.

**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.
```

**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 (leftmost) 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 is:

```
1111 1111 1111 1100
```

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

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

The Pascal program, `arshft.p`

```pascal
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
```

The command `asl`.

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

\texttt{asl(num, sh)}

Arguments

\textit{num} and \textit{sh} are integer expressions.

Return Value

\texttt{asl} returns a 32-bit integer value.

Comments

\texttt{asl} shifts the bits in \textit{num} \textit{sh} places to the left. \texttt{asl} preserves the sign bit of \textit{num} and does not wrap bits from left to right.

The result \texttt{asl} returns is machine-dependent and is unspecified unless the following is true:

\[ 0 \leq \textit{sh} \leq 32 \]
Example

The Pascal program, asl.p

```pascal
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
```

The asr function is identical to the arshft function. See the arshft listing.
The `card` function returns the number of elements in a set variable.

**Syntax**

```pascal
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.

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`

```pascal
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)
close

The close procedure closes a file.

Syntax

\texttt{close(file)}

Arguments

\texttt{file} is a file having the \texttt{text} or \texttt{file} data type.

Return Value

\texttt{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 when it 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 can trap this error with the I/O error recovery mechanism, described in “I/O Error Recovery” on page 214.

In Pascal, you cannot 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.

concat

The `concat` function returns the concatenation of two strings.

Syntax

`concat(str1, str2)`

Arguments

`str1` is a variable-length string, a character array, or a character-string constant.

`str2` is a variable-length string, a character array, or a character-string constant.

Return Value

`concat` returns a variable-length string.
Comments

concat returns a concatenation of \textit{str1} and \textit{str2}. You can concatenate any combination of varying, array of char, constant strings, and single characters.

The string plus (+) operator returns the same result as the concat function.

If the resulting string is longer than the maximum length of the destination varying string, it is truncated to this maximum length. If the resulting string is longer than 65,535 characters, it is truncated to this length.

See also the section: “stradd.”

Example

The Pascal program, \texttt{concat.p} program\texttt{concat.example(output);}
var\begin{verbatim}
begin
  writeln(concat(color, 'bird' + '.'));
end.
\end{verbatim}

The commands to compile and execute \texttt{concat.p}

\begin{verbatim}
hostname% \texttt{pc \texttt{concat.p}}\texttt{hostname% \texttt{a.out}}
  Blackbird.
\end{verbatim}

date

The date procedure takes 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 that is 8 elements long for the "C" locale, or a variable-length string.

Return Value

date returns a character string in the form traditional for a given locale. For the "C" locale, the form is mm-dd-yy, where dd is the day, mm is the month, and yy is the year.

Comments

date puts a zero in front of the day and the year, so that they always consist of two digits.

Use the environment variable LC_TIME to set the necessary locale.

See also the section: “time.”

Example

The Pascal program, date.p

program date_example(output);

var
  s1: alfa;
  s2: array[1..8] of char;
  s3: array[89..96] 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.
discard

The `discard` procedure removes 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.
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.
```
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

```pascal
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.

```text
hostname% pc expo.p
hostname% a.out
```

```text
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  4.03428793492773e+02  2
7  1.09663315842846e+03  3
8  2.98095798704173e+03  3
9  8.10308392757538e+03  3
10  2.20264657948068e+04  4
```

filesize

The `filesize` function returns the size of a given file.
Syntax

filesize(file)

Arguments

file is a variable with the text or file data type.

Return Value

filesize returns an integer value.

Comments

The argument can be either a text file of text type, or a binary file of a certain file of T type. It must be associated with an open file, otherwise an error occurs.

For a text file, filesize returns the number of bytes in the file.

For a binary file of type file of T, filesize returns the number of elements of type T in the file.

See also the sections, “seek,” and “tell.”
Example

The Pascal program, \texttt{filesize.p}

```pascal
program filesize_example;
var
  ft: text;
  fi: file of integer;
  i: integer;
begin
  rewrite(ft);
  rewrite(fi);
  i := 10;
  write(ft, i, i);
  write(fi, i, i);
  writeln('size of a text of an integer = ', filesize(ft): 3, ' bytes');
  writeln('size of a file of an integer = ', filesize(fi): 3, ' elements');
  close(ft);
  close(fi)
end. { filesize_example }
```

The commands to compile and execute \texttt{filesize.p}

```
hostname% pc filesize.p
hostname% a.out
size of a text of an integer = 20 bytes
size of a file of an integer = 2 elements
```

\textbf{firstof}

The \texttt{firstof} function returns the value of the lower bound when its argument is or has an ordinal type. For array types, \texttt{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.

\textbf{Syntax}

\texttt{firstof(x)}
Arguments

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

Return Value

The return value depends on the type that $x$ is.

<table>
<thead>
<tr>
<th>When $x$ is ...</th>
<th>The value of $\text{firstof}$ returns ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>An ordinal type, a constant, an expression, or variable</td>
<td>Has the same data type as its argument.</td>
</tr>
<tr>
<td>An array</td>
<td>Has the same data type as the type of the array index.</td>
</tr>
<tr>
<td>A set type</td>
<td>Has the same data type as the base type of the set.</td>
</tr>
</tbody>
</table>

Comments

Pascal follows the rules in Table 6-9 when returning the value of $x$. 
Table 6-9  firstof Return Values

<table>
<thead>
<tr>
<th>Type of Argument</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer (without –xl option)</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>integer (with –xl option)</td>
<td>-32,768</td>
</tr>
<tr>
<td>integer16</td>
<td>-32,768</td>
</tr>
<tr>
<td>integer32</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>char</td>
<td>chr(0)</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>Enumerated</td>
<td>The first element in the enumeration type declaration.</td>
</tr>
<tr>
<td>array</td>
<td>The lower bound of the subrange that defines the array size.</td>
</tr>
<tr>
<td>varying</td>
<td>1</td>
</tr>
<tr>
<td>set of ‘A’...’Z’</td>
<td>A (the character A).</td>
</tr>
</tbody>
</table>

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.
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`:

```pascal
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.');
end; { read_file }
```

flush.p (Screen 1 of 2)
The `getenv` function returns the value associated with an environment name.

```pascal
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)

```bash
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.
```

The commands to compile and execute `flush.p`
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 Solaris documentation for a complete description of environment variables.
```pascal
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

call file returns a value of type univ_ptr.

Comments

You can use the result of call file 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 call file; 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 produce errors 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 }
```
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 can 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 `halt`.

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

The Pascal program, \texttt{halt.p}

```pascal
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 \texttt{halt.p}

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

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 leftmost 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

See the example that follows.
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;

{ main program }
begin
    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.
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, as shown in Table 6-10.

<table>
<thead>
<tr>
<th>Value of Bit in <code>int1</code></th>
<th>Value of Bit in <code>int2</code></th>
<th>Value of Bit in result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

```pascal
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#00000000000000000000000000001111;
  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.

```plaintext
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
```

### 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` cannot 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 \texttt{lastof} returns has the same data type as the base type of the set.

\textit{Comments}

Pascal follows the rules in Table 6-11 when returning the value of \( x \).

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Type of Argument & Return Value \\
\hline
\texttt{integer} (without -xl) & 2,147,483,647 \\
\texttt{integer} (with -xl) & 32,767 \\
\texttt{integer16} & 32,767 \\
\texttt{integer32} & 2,147,483,647 \\
\texttt{char} & \texttt{chr}(255) \\
\texttt{boolean} & \texttt{true} \\
\texttt{enumerated} & The last element in the enumeration type declaration. \\
\texttt{array} & The upper bound of the subrange that defines the array size. \\
\texttt{varying} & The upper bound of the \texttt{varying} string. \\
\texttt{set of 'A'..'Z'} & The character \texttt{Z}. \\
\hline
\end{tabular}
\caption{\texttt{lastof} Return Values}
\end{table}

\textit{Example}

See the example under \texttt{firstof} on page 126.

\texttt{length}

The \texttt{length} function returns the length of a string.

\textit{Syntax}

\texttt{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  

```
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.
```
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

```pascal
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 }
```
### 6

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

```plaintext
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
Bria

linelimit.dat : Line limit exceeded
*** a.out terminated by signal 5: SIGTRAP
*** Traceback being written to a.out.trace
Abort (core dumped)
```

### lnot

The `lnot` function returns the bitwise `not` of an integer value.

#### Syntax

```plaintext
lnot(int)
```

#### Arguments

`int` is an integer expression.

#### Return Value

`lnot` returns an integer value.

#### Comments

`lnot` performs a bit-by-bit `not` operation, as shown in Table 6-12.
Table 6-12  lnot Truth

<table>
<thead>
<tr>
<th>Value of Bit in int</th>
<th>Value of Bit in result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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 under land on page 142.

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, as shown in Table 6-13.
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 under \( land \) on page 142.

\[ lshft \]

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.

The Pascal program, lshft.p

```pascal
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
```

**lsl**

The `lsl` function is identical to the `lshft` function. See the `lshft` listing on page 151.

**lsr**

The `lsr` function is identical to the `rshft` function. See the `rshft` listing on page 171.

**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 on page 156.

**Example**

```pascal
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 Pascal program `max.p`
message

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\((x1, \ldots, xN)\) is equivalent to the following code:

```plaintext
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.
```
Example

The Pascal program, message.p

```pascal
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.
```

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
\texttt{min} returns the same or the converted type of \texttt{exp1} and \texttt{exp2}.

Comments
See also the \texttt{max} listing on page 153.

Example
See the example under the \texttt{max} listing on page 153.

\texttt{null}

The \texttt{null} procedure performs no operation.

Syntax
\texttt{null}

Arguments
\texttt{null} does not take any arguments.

Return Value
\texttt{null} does not return any values.

Comments
\texttt{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 \texttt{case} statement, you could put \texttt{null} in place of the \texttt{case} statement, then replace it later 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

open takes the following 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 path name.
- A string of '^n', where n is an integer from 1 to 9. n represents the nth 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.'
- A variable or constant that contains any of the above items.
history instructs the compiler whether to create the file or what to do with it if it exists. history must be one of these values:

- 'new' Associates the operating system file with a new file. The compiler generates an error if the file already exists.
- 'old' Associates the operating system file with an existing file. The compiler generates an error if the file does not exist. This option first tries to open the file for writing. Failing to do so, it tries to open it for reading only.
- 'unknown' Searches for an existing file and associate it with the operating system file. The compiler creates the file if it does not exist.

Pascal returns an integer error code through error, as shown in Table 6-14.

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>open is successful.</td>
</tr>
</tbody>
</table>
| 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.
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-- ';

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

```plaintext
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.
```

**pcexit**

The `pcexit` function:

- Checks whether any imposed statement count has been exceeded.
- Calls the `ieee_retrospective()` routine. See the Solaris documentation for details.
- Terminates the program with the specified return value (similar to the C `exit()` function).

**Syntax**

```plaintext
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 reverse order of their registration.
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 on page 172 to reseed the number generator.
Example

The Pascal program, random.p

```pascal
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
```

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

```pascal
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* is 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* picks up where the last *read* left off. With *readln*, the rest of the line is discarded, 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 error message:

Unknown name "value" found on enumerated type read
Trace/BPT trap (core dumped)

You can trap this error with the I/O error recovery mechanism, described in “I/O Error Recovery” on page 214. 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**

The Pascal program, `read.p`

```pascal
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.
```
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 can trap this error with the I/O error recovery mechanism, described in “I/O Error Recovery” on page 214.
Example

The Pascal program, remove.p

```pascal
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

```bash
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
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* gives you permission to read from the file, but not 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*:

```pascal
reset(data, 'primes');
```

*reset* does an implicit close on the file, hence you can reuse its file variable with a different file. Similarly, if *input* or *output* is 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 can trap this error with the I/O error recovery mechanism, described in “I/O Error Recovery” on page 214.

See also the section on “*rewrite,*” which opens a file for writing.

Example

See the example in the *rewrite* listing that follows.

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` gives you permission 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, thus you can reuse its file variable with a different file.

See also the section on “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
    readin(f, line);
    writeln(f, 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.

**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 (leftmost) 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
```

**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 following statement:

x := seed(wallclock);

Example

See the example that follows.
The seek procedure changes the current position in the file.

**Syntax**

`seek(file, pos)`

**Arguments**

- `file` is a variable with the text or file data type.
- `pos` is a positive integer.

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

seek

The seek procedure changes the current position in the file.
Return Value

seek does not return any values.

Comments

The seek procedure is a facility to support random access input/output. It changes the position of a given file that is open for reading or writing.

You can use seek with text files of text type, or binary files of a certain file of T type.

For a binary file of type file of T, the argument pos denotes the number of the element of type T, which becomes the new position of file. Elements are numbered from 0. The argument pos can have an arbitrary non-negative value.

If file is open for writing, and pos exceeds the size of the file, the file is appended by the corresponding number of elements of type T with undefined values. For example, if filesize(f) = 0, then after seek(f,100) and write(f,x), the result is: filesize(f) = 101.

If file is open for reading, seek does not detect an error in seeking an element with a non-existing number. The compiler may detect this error later, however, when it performs the read procedure.

For a text file, you can use seek only in the following forms:

seek(file, 0) or seek(file, tell(file))

That is, seek can only set the current position to the beginning of the file or have it stay “as is,” otherwise an error occurs. Hence, the only correct way of processing a text file in Pascal is reading or writing it successively, line by line.

See also the sections: “filesize,” and “tell.”
Example

The Pascal program, seek.p

```pascal
program seek_example;
var
  f: file of integer;
  i: integer;
begin
  rewrite(f);
  for i:= 0 to 9 do
    write(f, i);
  writeln('Initial size of f =', filesize(f) :3, ' elements');
  reset(f);
  seek(f, 4);
  read(f, i);
  writeln('The 4th element of f =', i :3);
  rewrite(f);
  write(f, i);
  seek(f, 100);
  write(f, i);
  writeln('Final size of f =', filesize(f):3, ' elements');
  close(f);
end. { seek_example }
```

The commands to compile and execute seek.p

```
hostname% pc seek.p
hostname% a.out
Initial size of f = 10 elements
The 4th element of f = 4
Final size of f =101 elements
```

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. This argument is optional.

Return Value

`sizeof` returns an integer value.

Comments

`sizeof` returns the number of bytes in the data object $x$. $tags$ correspond to the fields in a variant record. $tags$ 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 runtime. The difference between the size of a constant string and that of a varying string variable to which the string is assigned. For example:

```
sizeof ('') = 0
```

However, if $S$ is defined as follows:

``` pascal
var S: varying [12] of char;
begin
  S:='';
end;
```

then `sizeof (S) = 16`. Moreover,

```
sizeof ('''+)''') = 4
```

because the `+' string operator returns a varying string object.
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 is ', sizeof(rec.ar), '.');
  writeln('The size of subB is     ', sizeof(subB), '.');
  writeln('The size of sub1 (8) is  ', sizeof(subl), '.');
  writeln('The size of sub2 (128) is ', sizeof(sub2), '.');
  writeln('The size of sub3 (256) is ', sizeof(sub3), '.');
  writeln('The size of univ_ptr is ', sizeof(u), '.')
end. {sizeof_example}
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` is 1.
The size of `char` is 1.
The size of `color1` is 1.
The size of `color2` is 1.
The size of `file` is 2089.
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` is 20.
The size of `subB` is 1.
The size of `sub1 (8)` is 1.
The size of `sub2 (128)` is 1.
The size of `sub3 (256)` is 2.
The size of `univ_ptr` is 4.

### 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 code in your source program:

```pascal
{$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.

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`

```pascal
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)
```

**stradd**

The `stradd` procedure adds a string to another string.

**Syntax**

```
stradd(strdest, srcstr)
```

**Arguments**

- `strdest` is a variable-length string.
- `strsrc` is a variable-length string, a character array, or a character-string constant.

**Return Value**

`stradd` does not return any values.

**Comments**

`stradd` adds `strsrc` to `strdest`, and is a more efficient operator for the concatenation of strings. Use `stradd` when a string is constructed by multiple concatenation, with other strings to its end.

`stradd` avoids allocating temporary storage. An example is the assignment `str1 := str1 + str2`, in which the compiler copies `str1` into some temporary storage, appends `str2`, and copies the result back into `str1`. 
If the resulting string is longer than the maximum length of `strdest`, it is truncated to this maximum length.

See also the section: “concat.”

**Example**

```pascal
The Pascal program, stradd.p

```program stradd_example(output);
var
begin
  stradd(greeting, ',');
  stradd(greeting, ' world');
  stradd(greeting, '!');
  writeln(greeting);
end.
```

The commands to compile and execute stradd.p

```
hostname% pc stradd.p
hostname% a.out
Hello, world!
```

**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 or zero.
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`:

```pascal
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
```

`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.

tell

The tell function returns the current position of a given file.

**Syntax**

tell(file)

**Arguments**

file is a variable with the text or file data type.

**Return Value**

tell returns an integer value.
Comments

The argument can be either a text file of text type, or a binary file of a certain file of T type. It must be associated with an open file, otherwise an error occurs.

For a text file, the tell function returns the byte number that corresponds to the current position in the file.

For a binary file of type file of T, the tell function returns the number of the element of type T that corresponds to the current position in the file. Elements are numbered from 0.

See also the sections on: “filesize,” “seek.”

Example

```
program tell_example;
var
  ft: text;
  fi: file of integer;
  i: integer;
begin
  rewrite(ft);
  rewrite(fi);
  for i:= 1 to 3 do begin
    writeln('tell(ft) =', tell(ft) :3);
    writeln('tell(fi) =', tell(fi) :3);
    writeln(ft, i :3);
    write(fi, i);
  end;
  close(ft);
  close(fi)
end. { tell_example }
```
The time procedure retrieves the current time.

Syntax

time(a)

Arguments

a is a variable that can be either a character array that is 8 elements long for the "C" locale, or a variable-length string.

Return Value

time returns a character string in the form traditional for a given locale. For the "C" locale, the form is hh:mm:ss, where hh is the hour (0 through 23); mm is the minutes (0 through 59); and ss is the seconds (0 through 59).

Comments

time uses a 24-hour clock. It puts a zero in front of the hours, minutes, and seconds, so that they always consist of two digits.

Use the environment variable LC_TIME to set the necessary locale.

See also the section: “date.”
Example

The Pascal program, time.p

```pascal
program time_example(output);

var
  s1: alfa;
  s2: array[1..8] of char;
  s3: array[89..96] 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.
```
The commands to compile and execute `time.p`:

```
hostname% pc time.p
hostname% a.out
The time is 14:02:49.
The time is 14:02:49.
The time is 14:02:49.
The time is 14:02:49.
hostname% setenv LC_TIME ru
hostname% a.out
The time is 14:02:56.
The time is 14:02:56.
The time is 14:02:56.
The time is 14:02:56.
hostname% setenv LC_TIME C
hostname% a.out
The time is 14:03:21.
The time is 14:03:21.
The time is 14:03:21.
The time is 14:03:21.
```

`trace`

The `trace` routine prints stack traceback without terminating a program.

**Syntax**

```
trace
```

**Arguments**

`trace` does not take any arguments.

**Return Value**

`trace` does not return any values.
Comments

You can use the `trace` routine for debugging. This routine prints stack traceback information to a file without terminating your program. The name of the traceback file is `p.trace`, where `p` is the name of your executable. For example, if the executable is called `a.out`, then the name of the traceback file is `a.out.trace`.

The `trace` routine can be called several times during program execution, if necessary. In this case, traceback information is appended to the traceback file.

The `trace` routine uses `dbx`, so be sure that `dbx` is in your path.

To print the traceback output in a clearer format, use the `-g` option to compile your program.

Example

The Pascal program `trace.p`

```pascal
program trace_example;

procedure subr(count: integer);
begin
  if (count > 0 ) then
    subr(count - 1)
  else
    trace;
end;

begin
  subr(5);
end.
```
The commands to compile and execute trace.p

```
hostname% pc trace.p -g
hostname% a.out
hostname% cat a.out.trace
```

### Stacktrace of a.out

```
[4] subr(count = 0), line 8 in "trace.p"
[5] subr(count = 1), line 6 in "trace.p"
[6] subr(count = 2), line 6 in "trace.p"
[7] subr(count = 3), line 6 in "trace.p"
[8] subr(count = 4), line 6 in "trace.p"
[9] subr(count = 5), line 6 in "trace.p"
[10] program(), line 12 in "trace.p"
detaching from process 28226
```

**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

The Pascal program, trim.p

```pascal
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
0123401234567891234560123456789         31
```
Type Transfer

The type transfer function changes 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:

```pascal
var
  x: integer32;
  y: single;
```

To transfer the value of variable `x` to a floating-point number, you would write:

```pascal
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, a 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**

```pascal
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 }
```
wallclock

The `wallclock` function returns the elapsed number of seconds since 00:00:00 GMT January 1, 1970.

**Syntax**

```plaintext
wallclock
```
Arguments

wallclock does not take any arguments.

Return Value

wallclock returns an integer value.

Comments

wallclock can be used with the seed function to generate a random number. It can also be used to time programs or parts of programs.

Example

See the example that follows.
The Pascal program, \texttt{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.')
nend. { wallclock_example }
```

The commands to compile and execute \texttt{wallclock.p}

```
hostname% pc wallclock.p
hostname% a.out
Begin computing fibonacci series.
  20 Iterations: Fibonacci(24) = 46368.
  Elapsed time is  5 seconds.
```
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.

Return Value

write and writeln do not return any values.

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. It can also be a constant, expression, or variable.
To write an expression in hexadecimal, use this form:

```pascal
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**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Width without <code>-xl</code> Option</th>
<th>Default Width with <code>-xl</code> Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>array of char</td>
<td>Declared length of the array</td>
<td>Declared length of the array</td>
</tr>
<tr>
<td>boolean</td>
<td>Length of <code>true</code> or <code>false</code></td>
<td>15</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>double</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>enumerated</td>
<td>Length of type</td>
<td>15</td>
</tr>
<tr>
<td>hexadecimal</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>integer</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>integer16</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>integer32</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>longreal</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>octal</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>real</td>
<td>21</td>
<td>13</td>
</tr>
<tr>
<td>shortreal</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>single</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>string constant</td>
<td>Number of characters in string</td>
<td>Number of characters in string</td>
</tr>
<tr>
<td>variable-length string</td>
<td>Current length of the string</td>
<td>Current length of the string</td>
</tr>
</tbody>
</table>
Example

The Pascal program `octal.p`

```pascal
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.
```

`xor`

The `xor` function returns the exclusive or of two integer values.

**Syntax**

```pascal
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>
<thead>
<tr>
<th>Value of Bit in int1</th>
<th>Value of Bit in int2</th>
<th>Value of Bit in result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

If int1 and int2 are different size integers, Pascal converts the smaller integer to the larger integer before 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 on page 142.
This chapter describes the Pascal input and output environments, with emphasis on interactive programming. It contains the following sections:

<table>
<thead>
<tr>
<th>Input and Output Routines</th>
<th>page 203</th>
</tr>
</thead>
<tbody>
<tr>
<td>eof and eoln Functions</td>
<td>page 204</td>
</tr>
<tr>
<td>More About eoln</td>
<td>page 208</td>
</tr>
<tr>
<td>External Files and Pascal File Variables</td>
<td>page 210</td>
</tr>
<tr>
<td>input, output, and errout Variables</td>
<td>page 211</td>
</tr>
<tr>
<td>Pascal I/O Library</td>
<td>page 213</td>
</tr>
<tr>
<td>Buffering of File Output</td>
<td>page 213</td>
</tr>
<tr>
<td>I/O Error Recovery</td>
<td>page 214</td>
</tr>
</tbody>
</table>

### Input and Output Routines

Pascal supports all standard input and 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.”
A 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, the system cannot detect 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 runtime action by you. 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 must type some input before execution can begin. This makes 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`. If you force Pascal to determine whether the input is at the end of the file or the end of the line, it must attempt to read from the input.

Consider the following example:
At first glance, this may appear to be a correct program for requesting, reading, and echoing numbers. However, the while loop asks whether `eof` is true before the request is printed. Thus, this system is forced to decide whether the input is at the end of the file. It gives no messages; it simply waits for the user to type a line, as follows:

```pascal
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 }
```

The commands to compile and execute `eof_example1.p`:

```bash
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)
```

The following code avoids this problem by prompting before testing `eof`:
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, because 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:

```
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)
```

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:

```
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 )
```
In general, unless you test the end-of-file condition both before and after calls to \texttt{read} or \texttt{readln}, there may be input that causes your program to attempt to read past the end-of-file.

\textbf{More About eoln}

To have a good understanding of when \texttt{eoln} is \texttt{true}, remember that in any file text, there is a special character indicating end-of-line. In effect, Pascal always reads one character ahead of the \texttt{read} command.

For instance, in response to \texttt{read(ch)}, Pascal sets \texttt{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:

```
read(ch);
if eoln then
    Done with line
else
    Normal processing
```

This program 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)`.

```
read(ch);
if eoln then
    Done with line
else begin
    read(ch);
    Normal processing
end
```

Given this framework, the function of a `readln` call is defined as follows:

```
while not eoln do
    get(input);
get(input);
```

This code 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.
External Files and 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>
<thead>
<tr>
<th>Association</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>With the open function</td>
<td><code>open</code> associates a permanent file with a file variable for reading or writing. <code>open</code> can also determine if a file actually exists.</td>
</tr>
<tr>
<td>With the reset and rewrite functions</td>
<td>In Pascal, <code>reset</code> and <code>rewrite</code> 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.</td>
</tr>
<tr>
<td>With the program header</td>
<td>If you call <code>reset</code> or <code>rewrite</code> with a file variable <code>f1</code>, which is bound to a file variable declared <code>f2</code> in the program header and do not specify the file name, Pascal opens a file with the same name as the variable <code>f2</code>. <code>reset</code> gives a runtime error if the file does not exist. <code>rewrite</code> creates the file if it does not exist.</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Association</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>With the procedure:</td>
<td>file_variable must not be declared in the program statement. This procedure creates a temporary file called <code>#tmp.suffix</code>, where <code>suffix</code> is unique to that temporary file. When the program exits or leaves the scope in which file_variable is declared, the file is deleted.</td>
</tr>
<tr>
<td>rewrite(file_variable)</td>
<td></td>
</tr>
<tr>
<td>With the procedure:</td>
<td>The procedure creates the temporary file called <code>#tmp.suffix</code>, where <code>suffix</code> is unique to that temporary file. This file is not deleted after program execution.</td>
</tr>
<tr>
<td>rewrite(output)</td>
<td></td>
</tr>
</tbody>
</table>

input, output, and errout Variables

The input, output, and errout variables are special predefined file variables.

- input is equivalent to the operating system standard input file, stdin.
- output is equivalent to the operating system standard output file, stdout.
- errout is equivalent to the 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 filename and associates it with input. read and readln read from that file. For example, this line opens the file, some/existing/file, and associates it with input:

```
reset(input,'some/existing/file');
```

You must supply a file name for the association to work.

**Associating output with a File Other Than stdout**

To associate output with a file other than stdout, call rewrite(output, filename). Pascal opens 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.

**Associating errout with a File Other Than stderr**

To associate errout with a file other than stderr, call:

```
rewrite (errout, '/some/new/file');
```

Subsequently, whenever you direct write or writeln to errout, the output is sent to /some/new/file. You obtain the same results when you write a string to errout implicitly, using the message function. See “message” on page 155 for details.
Pascal I/O Library

Each file variable in Pascal is associated with a data structure. The data structure defines the physical Solaris 2.x 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 the output that has been generated for output whenever one of the following conditions 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:
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 1,024. You can specify this option in a program comment using one of these formats:

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

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 is used for the entire program.

**I/O Error Recovery**

When an I/O routine encounters an error, it normally does the following:

1. Generates an error message.
2. Flushes its buffers.
3. Terminates with a SIGTRAP.

Although you can set up a signal handler to trap this signal, you cannot determine which routine called the signal or the reason it was called.

With Pascal, you can 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 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 outermost 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 must 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 different directories for Solaris 1.x and Solaris 2.x:

Solaris 1.x /usr/lang/SC4.0/include/pascal
Solaris 2.x /opt/SUNWspro/SC4.0/include/pascal

If the compiler is installed in a non-default location, change `/usr/lang` or `/opt/SUNWspro` 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
{$w-}
program ioerr_example(output);
{ This program sets and uses an I/O trap routine. }

#include "ioerr.h"

c
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
```
Overview of Pascal Extensions

This Appendix gives an overview of the Pascal extensions to ISO/ANSI standard Pascal.

Lexical Elements

Pascal supports the following extensions to the lexical elements of standard Pascal:

- Uppercase- and lowercase-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 A-1
- An underscore (_) and dollar_sign($) in identifier names
- The comment delimiters /* */. in addition to the standard (* *) and ({ }
- The comment delimiters " "

Table A-1  Nonstandard Identifiers

<table>
<thead>
<tr>
<th>Nonstandard Identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE      close index lsr  return</td>
</tr>
<tr>
<td>TRUE       concat integer16 max rshft</td>
</tr>
<tr>
<td>addr       date integer32 maxchar seed</td>
</tr>
<tr>
<td>alfa       discard intset message seek</td>
</tr>
<tr>
<td>append     double land min shortreal</td>
</tr>
<tr>
<td>argc       exit lastof minchar single</td>
</tr>
<tr>
<td>argv       expo length minint sizeof</td>
</tr>
<tr>
<td>arshft     filesize linelimit next stradd</td>
</tr>
<tr>
<td>asl        firstof lnot null substr</td>
</tr>
<tr>
<td>asr        flush longreal open tell</td>
</tr>
<tr>
<td>assert     getfile lor pack trace</td>
</tr>
<tr>
<td>bell       getenv lshft random trim</td>
</tr>
<tr>
<td>card       halt lsl remove univ_ptr</td>
</tr>
<tr>
<td>clock      in_range</td>
</tr>
</tbody>
</table>

**Data Types**

Pascal supports the following 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-length 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 the following 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

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
- Functions returning structured types

Built-In Routines

Pascal supports the following nonstandard built-in routines:

- The addr function, which returns the address of a specified variable
- The append procedure, which opens a file for writing at its end
- 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 `concat` function, which concatenates two strings
• 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 `filesize` function, which returns the current size of a file
• 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 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 Solaris 2.0 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 `seek` procedure, which resets the current position of a file
• The `sizeof` function, which returns the size of a specified type, variable, constant, or string
• The `stlimit` procedure, which terminates program execution if a specified number of statements have been executed in the current loop
• The `stradd` procedure, which adds a string to the end of another string
• The `substr` function, which extracts a substring from a string
• The `sysclock` function, which returns the system time used by the process
• The `tell` function, which returns the current position of a file
• The `time` procedure, which retrieves the current time
• The trace procedure, which prints stack traceback
• The trim function, which removes the trailing blanks in a character string
• The type_transfer function, which changes 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:
• Association of a Pascal file with either a permanent or temporary Solaris operating system file
• The special predefined file variables, input and output, that 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:
• Sharing variable, procedure, and function declarations across multiple units using include files
• Sharing variable, procedure, and function declarations across multiple units using multiple declarations
• Sharing variable, procedure, and function declarations across multiple units using the extern and define variable declarations
• Sharing variable, procedure, and function declarations between units of different languages using the extern and external routine options
This Appendix describes the differences between Pascal and Apollo DOMAIN Pascal, and how the \(-x1\) option can be used to get around most of these differences.

**The \(-x1\) Option**

The \(-x1\) option to the `pc` command makes the language accepted by the Pascal compiler similar to DOMAIN Pascal. Table B-1 lists the differences in your program when you compile it with and without the \(-x1\) option.

<table>
<thead>
<tr>
<th>With (-x1)</th>
<th>Without (-x1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The default <em>integer</em> size is 16 bits.</td>
<td>The default is 32 bits.</td>
</tr>
<tr>
<td>The default <em>real</em> size is 32 bits.</td>
<td>The default is 64 bits.</td>
</tr>
<tr>
<td>The default enumerated type size is 16 bits.</td>
<td>The default is either 8 or 16 bits, depending on the number of elements in the enumerated set.</td>
</tr>
<tr>
<td>The source file is run through the preprocessor <code>cppas</code> before it is processed by the compiler.</td>
<td>The source file is run through the preprocessor <code>cpp</code>.</td>
</tr>
<tr>
<td>Pascal supports <code>nonpascal</code> as a routine option.</td>
<td><code>nonpascal</code> is not supported.</td>
</tr>
</tbody>
</table>
Table B-1  Differences Between Programs Compiled with and without –xl  (Continued)

<table>
<thead>
<tr>
<th>With –xl</th>
<th>Without –xl</th>
</tr>
</thead>
<tbody>
<tr>
<td>The –L option, which maps all identifiers to lowercase, is on by default.</td>
<td>–L is off by default.</td>
</tr>
<tr>
<td>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.</td>
<td>The compiler generates an error and halts.</td>
</tr>
</tbody>
</table>
| The writing of enumerated and boolean variables defaults to uppercase 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. | These two types of modules are not linked 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 the following 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`
Implementation Restrictions

This Appendix describes the Pascal features that are implementation-defined.

Identifiers

Pascal restricts the maximum length of an identifier to 1,024 characters. All characters are significant.

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
real

Table C-1 lists the minimum and maximum values Pascal assigns to the real data types, single and double.

Table C-1  Values for single and double

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Maximum Value</th>
<th>Minimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>32</td>
<td>3.402823e+38</td>
<td>1.401298e-45</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>1.79769313486231470e+308</td>
<td>4.94065645841246544e-324</td>
</tr>
</tbody>
</table>

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 C-2.

Table C-2  maxint and minint

<table>
<thead>
<tr>
<th>Option</th>
<th>maxint</th>
<th>minint</th>
</tr>
</thead>
<tbody>
<tr>
<td>–xl off</td>
<td>2,147,483,647</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>–xl on</td>
<td>32,767</td>
<td>-32,768</td>
</tr>
</tbody>
</table>

Character

Pascal defines the maximum range of characters as 0 to 255.

Record

Pascal restricts the maximum size of a record to 2,147,483,647 bytes.

Array

Pascal restricts the maximum size of an array to 2,147,483,647 bytes.
Set

Pascal restricts the maximum size of a set to 32,767 elements.

Alignment

The size and alignment of data types depends on whether or not you compile your program with the \(-x1\) option. Table C-3 shows the representation of data types without \(-x1\), and Table C-4 shows the representation with \(-x1\).
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>Four bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>integer16</td>
<td>Two bytes</td>
<td>Two bytes</td>
</tr>
<tr>
<td>integer32</td>
<td>Four bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>real</td>
<td>Eight bytes</td>
<td>Eight bytes</td>
</tr>
<tr>
<td>single</td>
<td>Four bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>shortreal</td>
<td>Four bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>double</td>
<td>Eight bytes</td>
<td>Eight bytes</td>
</tr>
<tr>
<td>longreal</td>
<td>Eight bytes</td>
<td>Eight bytes</td>
</tr>
<tr>
<td>boolean</td>
<td>One byte</td>
<td>One byte</td>
</tr>
<tr>
<td>char</td>
<td>One byte</td>
<td>One byte</td>
</tr>
<tr>
<td>enumerated</td>
<td>One or two bytes, depending on the number of elements in the enumerated set</td>
<td>One or two bytes</td>
</tr>
<tr>
<td>subrange</td>
<td>One, two, or four bytes</td>
<td>One, two, or four bytes</td>
</tr>
<tr>
<td>record</td>
<td>Depends upon the base type of that field.</td>
<td>Four bytes</td>
</tr>
<tr>
<td>array</td>
<td>Requires the same space required by the base type of the array.</td>
<td>Same as element type</td>
</tr>
<tr>
<td>set</td>
<td>Pascal implements vector, with one bit representing each element of a set. The size is determined by the size of the ordinal value of the maximal element of the set plus one. It is a minimum of two bytes and always in two-byte multiples.</td>
<td>Two bytes if size = 2; otherwise, four bytes</td>
</tr>
<tr>
<td>pointer</td>
<td>Four bytes</td>
<td>Four bytes</td>
</tr>
</tbody>
</table>
Table C-4  Internal Representation of Data Types with \( \times 1 \)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>Two bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>integer16</td>
<td>Two bytes</td>
<td>Two bytes</td>
</tr>
<tr>
<td>integer32</td>
<td>Four bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>real</td>
<td>Four bytes</td>
<td>Eight bytes</td>
</tr>
<tr>
<td>single</td>
<td>Four bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>shortreal</td>
<td>Four bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>double</td>
<td>Eight bytes</td>
<td>Eight bytes</td>
</tr>
<tr>
<td>longreal</td>
<td>Eight bytes</td>
<td>Eight bytes</td>
</tr>
<tr>
<td>boolean</td>
<td>One byte</td>
<td>One byte</td>
</tr>
<tr>
<td>char</td>
<td>One byte</td>
<td>One byte</td>
</tr>
<tr>
<td>enumerated</td>
<td>Two bytes</td>
<td>Two bytes</td>
</tr>
<tr>
<td>subrange</td>
<td>Two or four bytes</td>
<td>Two or four bytes</td>
</tr>
<tr>
<td>record</td>
<td>Depends on the base type of that field.</td>
<td>Four bytes</td>
</tr>
<tr>
<td>array</td>
<td>Needs the same space required by the base type of the array.</td>
<td>Same as element type</td>
</tr>
<tr>
<td>set</td>
<td>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. It is a minimum of two bytes and always in two-byte multiples.</td>
<td>Two bytes if size = 2; otherwise, four bytes</td>
</tr>
<tr>
<td>pointer</td>
<td>Four bytes</td>
<td>Four bytes</td>
</tr>
</tbody>
</table>

**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 C-5 if you do not specify the minimum field length of a parameter.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Default Width without <code>-xl</code> Option</th>
<th>Default Width with <code>-xl</code> Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>array of char</td>
<td>Declared length of the array</td>
<td>Declared length of the array</td>
</tr>
<tr>
<td>boolean</td>
<td>Length of <code>true</code> or <code>false</code></td>
<td>15</td>
</tr>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>double</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>enumerated</td>
<td>Length of type</td>
<td>15</td>
</tr>
<tr>
<td>hexadecimal</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>integer</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>integer16</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>integer32</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>longreal</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>octal</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>real</td>
<td>21</td>
<td>13</td>
</tr>
<tr>
<td>shortreal</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>single</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>string constant</td>
<td>Number of characters in the string</td>
<td>Number of characters in the string</td>
</tr>
<tr>
<td>variable-length string</td>
<td>Current length of the string</td>
<td>Current length of the string</td>
</tr>
</tbody>
</table>
Pascal Validation Summary Report

The Pascal Version 3.0.2 compiler has been validated using Version 5.1 of the Pascal Validation Suite. It complies with FIPS PUB 109 ANSI/IEEE 770 X3.97-1983 and BS6192/ISO7185 at both level 0 and level 1. This appendix is a summary of the validation.

Test Conditions

The Pascal Version 3.0.2 compiler was validated under the Solaris 2.3 operating system on a SPARCstation™ 10 machine.

The following compiler options were used during each validation:

- Level 1 mode
- All checks
- Runtime trace
- All other default options

The following manufacturer’s statement of compliance is included in the Validation Summary Report for the architecture.

Manufacturer’s Statement of Compliance

The above processor complies with the requirements of both level 0 and level 1 (by means of a compiler switch) of BS 6192/ISO 7185, with no exceptions.
Implementation-Defined Features

The implementation-defined features are as follows:

E.1 The value of each char-type corresponding to each allowed string-character is the corresponding ISO 8859/1 (ASCII) character.

E.2 The subset of real numbers denoted by signed-real are the values representable in the single precision (32-bit) format of the IEC559:1982 Standard Binary Floating Point Arithmetic for Microprocessor Systems, which is the same format as in the IEEE standard P754.

E.3 The values of char-type are the ISO 8859/1 (ASCII) character set.

E.4 The ordinal numbers of each value of char-type are the corresponding ISO 8859/1 (ASCII) code value.

E.5 All file operations are performed at the point where they are encountered at execution time, with the exception of get (both explicit and where implied by reset and read), which is delayed in its execution to the point at which the file is next referenced—a technique known as “lazy I/O.”

E.6 The value of maxint is 2,147,483,647.

E.7 The accuracy of the approximations of the real operations and functions is determined by the representation (see E.2) and by the rounding of intermediate results. This gives approximately 16-decimal digits of precision.

E.8 The default value of TotalWidth for integer-type is 10.

E.9 The default value of TotalWidth for real-type is 21.

E.10 The default value of TotalWidth for boolean-type is 5.

E.11 The value of ExpDigits is 2.

E.12 The exponent character is e.

E.13 The case in the output of the value of boolean-type is uppercase for the initial letter, and lowercase for the remaining letters.
E.14 The procedure \texttt{Page} causes the contents of the output buffer (if any) to be written, and then outputs the ISO 8859/1 (ASCII) form-feed character. The effect on any device depends on that device.

E.15 There is no binding between physical files and program parameters of file-type. Variables of file-type are associated with physical files or devices automatically by the processor.

E.16 The effects of \texttt{reset} and \texttt{rewrite} on the standard files \texttt{input} and \texttt{output} depend on the binding of these files specified at the invocation of the program. In general, \texttt{reset} and \texttt{rewrite} have the effects described in clause 6.6.5.2 of the Pascal Standard\textsuperscript{1} when \texttt{input} and \texttt{output} have been bound to permanent files. When the binding is to a device, \texttt{reset(input)} has no effect other than discarding any partially processed line. \texttt{rewrite(output)} terminates any partially complete line but has no other effect. \texttt{rewrite(input)} and \texttt{reset(output)} are treated as errors.

E.17 This implementation supports the alternative representation of symbols permitted by the Standard.

\textit{Reporting of Errors}

The following errors are detected prior to, or during, execution of a program:


The following errors are not, in general, reported:


Implementation-Dependent Features

Implementation-dependent features F.1 to F.11 of Pascal are treated as undetected errors.

Extensions

The processor does not contain any extensions to BS6192/ISO 7185. Such extensions must be enabled by means of a compiling option, not the subject of validation.
This glossary defines some general programming terms, as well as terms that are specific to Pascal.

| ~ |
| The bitwise not operator.

| ! |
| The bitwise or operator.

| \\# |
| A programming symbol that specifies an integer value in a base other than 10, includes a file in your program, or indicates a preprocessor command.

| % |
| A programming symbol used with the -xl option for special cppas directives.

| & |
| The bitwise and operator.

| 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.

append

A built-in procedure that opens a file for writing at its end.

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, provided that the –C option is specified.

bell

A predeclared character constant equal to char(7), which makes the terminal beep.

block buffering

Output buffering with a block size of 1,024.

card

A built-in function that returns the number of elements of a set variable.

clock

A built-in function that returns the user time consumed by the process.
close

A built-in procedure that closes a file.

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.

concat

A built-in function that concatenates two strings.

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.

cppas

The preprocessor that handles the Pascal conditional variables and compiler directives when the -xl option is specified.

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.

%debug

A compiler directive that works with the -cond compiler option. -cond instructs pc, the Pascal compiler, 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*.

**%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 the `%else` directive 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 the `%elseif` directive 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 the `%elseifdef` directive 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 the `%endif` directive 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 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 causes 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.

filesize

A built-in function that returns the current size of a file.

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 operating system file.
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.

halt
A built-in procedure that terminates program execution.

%if
A compiler directive. When the compiler encounters a %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 the %if directive must be compiled with the –xl option.

%ifdef
A compiler directive that determines whether or not a conditional variable in a
%var directive has been previously defined. Programs that contain the
%ifdef directive must be compiled with the –xl option.

I/O handler
A Pascal function that is 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 instructs 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 directive.

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 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].

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.

max

A built-in function that evaluates two scalar expressions 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.

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.

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 is used to pass values out of the routine.
<table>
<thead>
<tr>
<th><strong>output</strong></th>
<th>A special predefined file variable equivalent to the standard output file, stdout.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>private</strong></td>
<td>A variable, procedure, or function declaration that restricts its accessibility to the current compilation unit.</td>
</tr>
<tr>
<td><strong>procedure and function pointer</strong></td>
<td>A pointer that has the address of a procedure or function as its value.</td>
</tr>
<tr>
<td><strong>public</strong></td>
<td>A variable, procedure, or function declaration that is visible across multiple programs and modules.</td>
</tr>
<tr>
<td><strong>random</strong></td>
<td>A built-in function that generates a random number between 0.0 and 1.0.</td>
</tr>
<tr>
<td><strong>remove</strong></td>
<td>A built-in procedure that removes the specified file.</td>
</tr>
<tr>
<td><strong>return</strong></td>
<td>A statement used in a procedure or function to prematurely end the procedure or function.</td>
</tr>
<tr>
<td><strong>rshft</strong></td>
<td>A built-in function that does a logical right shift of an integer value.</td>
</tr>
<tr>
<td><strong>seed</strong></td>
<td>A built-in function that reseeds the random number generator.</td>
</tr>
<tr>
<td><strong>seek</strong></td>
<td>A built-in procedure that resets the current position of a file.</td>
</tr>
<tr>
<td><strong>shortreal</strong></td>
<td>A real data type that represents a 32-bit floating point number. Same as single.</td>
</tr>
<tr>
<td><strong>single</strong></td>
<td>A real data type that represents a 32-bit floating point number. Same as shortreal.</td>
</tr>
</tbody>
</table>
sizeof
A built-in function that returns the number of bytes the program uses to store a data object.

%slibrary
A compiler directive that directs 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 -xI option.

stradd
A built-in procedure that adds a string to the end of another string.

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.

sysclock
A built-in function that returns the system time consumed by the process.

tab
A predeclared character constant equal to char(9), which makes a tab character.

tell
A built-in function that returns the current position of a file.
<table>
<thead>
<tr>
<th><strong>time</strong></th>
<th>A built-in procedure that retrieves the current time.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>trace</strong></td>
<td>A built-in procedure that prints stack traceback.</td>
</tr>
<tr>
<td><strong>trim</strong></td>
<td>A built-in function that removes trailing blanks in a character string.</td>
</tr>
<tr>
<td><strong>type transfer function</strong></td>
<td>A built-in function that changes the data type of a variable, constant, or expression.</td>
</tr>
<tr>
<td><strong>unit</strong></td>
<td>Either a program or a module.</td>
</tr>
<tr>
<td><strong>univ</strong></td>
<td>A modifier used before data types in formal parameter lists to turn off type checking for that parameter.</td>
</tr>
<tr>
<td><strong>univ_ptr</strong></td>
<td>See universal pointer.</td>
</tr>
<tr>
<td><strong>universal pointer</strong></td>
<td>A pointer used to compare a pointer of one type to another or to assign a pointer of one type to another.</td>
</tr>
<tr>
<td><strong>%var</strong></td>
<td>A compiler directive that defines conditional variables for the preprocessor. Programs that contain the %var directive must be compiled with the –xl option.</td>
</tr>
<tr>
<td><strong>variable attribute</strong></td>
<td>An attribute that determines how to allocate the variable. Variable attributes include static, extern, and define.</td>
</tr>
<tr>
<td><strong>variable initialization</strong></td>
<td>The initialization of a real, integer, boolean, character, set, record, array, or pointer variable in the var declaration of the program.</td>
</tr>
<tr>
<td><strong>variable routine option</strong></td>
<td>A routine option that is used to pass a routine a smaller number of actual arguments than the number of formal arguments defined in the routine.</td>
</tr>
</tbody>
</table>
variable scope

Either private or public. Visibility of a private variable is restricted to the current compilation unit. A public variable can 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.

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.

-xl option

An option of the pc command that causes the compiler to implement Pascal as DOMAIN Pascal.

xor

A built-in function that returns the exclusive or of two integers.
Index

A
addr function, 99, 222
alfa data type, 34
alignment of data types, 233
and operator, 2, 221
and then operator, 69 to 70
AnswerBook, xviii
Apollo DOMAIN Pascal, 227 to 229
append function, 102, 222
argc
function, 99, 105, 222
procedure, 105
argv procedure, 99, 105, 222
arithmetic
left shift, 109
operators, 66
right shift, 108
routines, 95
array data types, 34 to 37
alfa, 34
as function return value, 89
conformant, 89
data representation, 37
declaring variables, 34
initializing variables, 36
string, 34
univ parameter type, 89
varying, 34
arrays, 73
arshift function, 97, 107, 171, 222
ASCII character set, 1
asl function, 97, 109, 222
asr function, 97, 111, 171, 223
assert statement, 48, 221
assignment statement, 13, 18, 20, 22
assignments, 63 to 65
compatibility rules, 64
data types, 63
extensions, 221
null strings, 65
string constants, 65
strings, to and from, 64

B
–b option to pc command, 214
bell character, 23, 220
bit operators, 66
bitwise operators, 151
and, 2, 221
not, 201, 221
or, 2, 201, 221
block buffering, 214
boolean
expression, 48
operators, 66
    and then, 69 to 70
    or else, 71
boolean data types, 20 to 21
assignment compatibility rules, 63
declaring constants, 21
declaring variables, 20
initializing variables, 20
buffering
    block, 214
    file output, 213
    line, 214
built-in procedures and functions, 2
    nonstandard, 95 to 201, 203, 222
    standard, 95, 203

C
-C option to pc command, 48, 218
C programming language, 94, 204
card function, 96, 112, 223
case statement, 48, 51 to 52, 157, 221
    otherwise clause, 51, 58
    range of constants, 52
    with -xl option, 51
character
data type, 23
    assignment compatibility
        rules, 63
    bell, 23, 220
data representation, 23
declaring constants, 23
declaring variables, 22
maxchar, 23, 220
minchar, 23, 220
tab, 23, 220
set, 1
    string routines, 97
clock function, 99, 113, 223
close procedure, 98, 116, 204, 223
comments, 6, 48, 214, 219
concat function, 117, 223
conformant array, 89
const declaration, 77, 79, 222
conventions, typographical, xvii

D
data structure, 213
data types
    alignment, 233
    array, 34 to 37
    assignments, 63
    boolean, 20 to 22
    enumerated, 24
    extensions, 220
    file, 41
    integer, 232
    internal representation
        with -xl, 235
        without -xl, 234
    pointer, 41 to 45, 80
    real, 231
    real, 13 to 15
    record, 27 to 33
    set, 38 to 39
    size restrictions, 232
    space allocation, 232
    subrange, 25
date procedure, 99, 118, 223
declarations, 77 to 84
    const, 79, 222
    define, 222
    extensions, 221
    label, 222
    type, 222
    var, 14, 18, 27, 80 to 83
default field widths, 236
define
    declaration, 83, 222
    variable, 80, 222
discard procedure, 99, 120, 223
documentation, xviii to xix
DOMAIN Pascal, 227 to 229
    features accepted but ignored, 227, 228
    features not supported, 229
    -xl option, 227
double data type, 13, 220, 232

E

enumerated data, 23, 24
  assignment compatibility rules, 63
data representation, 24
  with read and readln
    procedures, 23
  with write and writeln
    procedures, 23
eof function, 135, 204 to 207
eoln function, 135, 204 to 209
error
  file, stderr, 212
  recovery of input and output, 214
errout file variable, 211
exit statement, 52 to 53, 221
expo function, 96, 123, 223
extensions, 219 to 225
  assignments and operators, 221
  built-in routines, 222
data types, 220
  heading and declarations, 221
  input and output, 225
lexicalex elements, 219
  procedures and functions, 222
  program compilation, 225
  statements, 221
extern
  option, 92, 222
  variable, 82, 84, 222
external option, See extern option

F

field widths, default, 236
file
  permanent, 210
  stderr, 211
  stdin, 212
  stdout, 211
  temporary, 211
file data type, 41
  with -s option, 41
  with -v0 and -v1 options, 41
file identifiers
  input, 117
  output, 117
file variable, 210
errout, 211
input, 211
output, 211
files
  external and Pascal file variables, 210
  how to close, 117
  permanent and temporary, 210
filesize function, 124, 223
firstof function, 96, 126, 223
flush procedure, 98, 130, 204, 213, 223
for statement, 52, 56, 221
formal parameter, 88, 92
FORTRAN programming language, 94
forward option, 91
function
  addr, 99
  append, 102
  argc, 99, 105
  arshft, 97, 107, 171
  asl, 97, 109
  asr, 97, 111, 171
  association with define
    declaration, 84
  built-in, 95
  card, 96, 112
  clock, 99, 113
  concat, 117
  declarations, 77
  eof, 135, 204 to 207
eoln, 135, 204 to 209
expo, 96, 123
extensions, 222
external option, 92
external option, 92
filesize, 124, 223
firstof, 96, 126
forward option, 92
getenv, 99, 132
getfile, 98, 134, 204, 215
in_range, 138
index, 97, 139
internal option, 92
land, 97, 142
lastof, 144
length, 97, 145
lnot, 97, 149
lor, 97, 150
lshft, 97, 152
ls1, 97, 153
lsr, 97, 153
max, 96, 153
min, 96, 156
nonpascal option, 94
parameters, 85 to 88
private, 84, 92, 222
public, 84, 222
random, 96, 162
return statement, 59 to 60
return value, 89, 121
returning structured types, 222
rshift, 97, 171
seed, 96, 172
sizeof, 96, 176
substr, 97, 183
sysclock, 99, 114, 184
tell, 224
time, 185
trim, 97, 191, 225
type transfer, 99, 193
var declaration, 14, 18, 21, 23, 28
variable option, 92
wallclock, 99, 195
xor, 97, 200

H
halt procedure, 99, 136, 223
headings
extensions, 221
function, 84
program, 210

I
identifiers, 2, 4, 54, 59
as labels, 77
in define declaration, 83
nonstandard predeclared, 4, 220
restrictions to, 231
standard predeclared, 4, 220
if statement, 48, 208, 221
implementation restrictions, 231 to 236
in out parameter, 85, 88
in parameter, 85, 88, 215
in_range function, 138, 223
include directive and statement, 216
index function, 97, 139, 223
initializing variables, 83
input
environment, 203 to 218
error recovery, 214
extensions, 225
file
stdin, 212
variable, 211
input and output
library, 212
routines, nonstandard and
standard, 204
trap handler, 215
integer data types, 16 to 20
assignment compatibility rules, 63
data representation, 19
declaring constants, 18
integer, 17
integer16, 17, 220
integer32, 220
maxint, 232
minint, 19, 232

G
g.getenv function, 99, 132, 223
g.getFile function, 98, 134, 204, 215, 223
global variable, 81
goto statement, 48, 54, 59, 221
exiting current block, 54
use of identifier in, 54
specifying in another base, 19
unsigned integer, 17
integer16, 20, 220
integer32, 17, 20, 220
interactive programming, 203
internal option, 92, 222
ioerr.h file, 215

K
keywords, 2

L
-L option to pc command, 2, 228
label declaration, 77, 222
land function, 97, 142, 223
lastof function, 144, 223
length function, 97, 145, 223
lexical
    characters, 1
    elements, 1
line buffering, 214
linelimit procedure, 98, 147, 204, 223
lnot function, 97, 149, 223
local variable, 80, 81, 100
longreal, 220
lor function, 97, 150, 224
lowercase characters, mapping, 2
lshft function, 97, 151, 224
lsl function, 97, 153
lsr function, 97, 153, 224

M
manuals, See documentation
mapping to lowercase characters, 2
max function, 153, 224
maxchar, 23, 220
message procedure, 98, 155, 204, 213, 224
min function, 96, 156, 224
minchar, 23, 220
mod operators, 66
modules
    extern or external option, 92, 222
    extern variables, 82, 84
    public and private routines, 84
    scope of variables, 80

N
nested routines, 235
next statement, 56
nil, 79
nonpascal option, 94, 222
nonstandard special symbols
    !, 2, 219
    #, 2, 219
    %, 2, 219
    &, 2, 219
    |, 2, 219
    ~, 2, 219
not operator, 2, 201, 221
null procedure, 99, 157, 224
null string assignments, 65

O
open procedure, 98, 117, 158, 204, 210, 224
operators, 66 to 76, 79
    and, 2, 221
    and then, 69 to 70
    arithmetic, 66
    bit, 66, 68
    boolean, 66, 68
    extensions, 221
    mod, 66
    not, 2, 201, 221
    or, 2, 201, 221
    or else, 70
    precedence of, 76
    relational, 66, 72
    set, 66, 71
    string, 66, 75
options for routines, 91 to 94
    extern or external, 92, 222
    internal, 92
nonpascal, 94, 227
variable, 92
or else operator, 70
or operator, 2, 201, 221
otherwise clause in case statement, 51, 58
out parameter, 85, 88
output
  environment, 203 to 218
  error recovery, 214
  extensions, 225
  file
    buffering, 213
    stdout, 212
    variable, 211

P
packed records, 30
parameters, 84 to 89
  formal, 86, 88, 92
  in, 85, 88, 215
  in out, 85, 88
  out, 85, 88
  passing conventions, 86
  type checking, 88
  univ type, 88
  value, 86
  var, 86, 88
Pascal
  extensions in the compiler, xiv
  symbols, 2
  validation summary, 237 to 240
pc command
  –b option, 214
  –C option, 48, 218
  document reference, 2
  –L option, 2, 228
  –s option, 2, 3
  –v0 option, 3
  –v1 option, 3
  –xl option, 9, 17, 24, 94, 227, 234
pcexit procedure, 99
pointer data type, 41 to 45
assignment compatibility rules, 63
data representation, 45
declaring variables, 42
initializing variables, 45
procedure and function, 43, 80
univ_ptr, 42
universal, 80
precedence of operators, 76
private
  function, 84
  procedure, 84
  variable, 81, 222
procedure
  append, 222
  argc, 105
  argv, 99, 105
  association with define declaration, 84
built-in, 95
close, 98, 116, 204
date, 99, 118
declarations, 77
discard, 99, 120
extensions, 222
extern option, 92
external option, 92
flush, 98, 130, 204, 213
forward option, 92
halt, 99, 136
internal option, 92
linelimit, 98, 147, 204
message, 98, 155, 204, 213
nonpascal option, 94
null, 99, 157
open, 98, 117, 158, 204, 210
parameters, 85 to 89
pcexit, 99
private, 92, 222
public, 222
read, 23, 98, 163, 204, 205, 206, 209, 211
readln, 23, 98, 135, 163, 204, 207, 209, 211
remove, 98, 166, 204
reset, 98, 117, 135, 167, 204, 212
Index

return statement, 59
rewrite, 98, 135, 168, 204, 211
seek, 174, 224
stlimit, 99, 180
stradd, 182, 224
time, 99, 187
trace, 225
var declaration, 14, 21, 23, 28, 36, 39, 83
variable option, 92
write, 24, 98, 155, 198, 204, 211, 212
writeln, 24, 98, 155, 198, 204, 211, 212
program
  compilation extensions, 225
  headings, 210
  unit, 84
public
  function, 84
  procedure, 84
  variable, 81, 84, 222

R
random function, 162, 224
read procedure, 23, 98, 163, 204, 205, 207, 209, 211, 224
readln procedure, 23, 98, 135, 163, 207, 209, 211, 224
real data types, 13 to 15
  as function return value, 89
  data representation, 15
  declaring
    constants, 14
    variables, 13
double, 13, 220, 232
longreal, 13, 220
real, 13, 15
shortreal, 13, 15, 220
single, 13, 15, 220, 232
with -xi option, 13, 17
record data type, 26 to 33
  as function return value, 89
  assignment compatibility rules, 63
  declaring variables, 26
  initializing
    data, 27
    variables, 83
    representation of unpacked records, 30
records, 73
relational operators, 66, 72
remove procedure, 98, 166, 204, 224
repeat statement, 52, 56, 221
reserved words, 3
  nonstandard extensions, 4
  standard, 3
reset procedure, 98, 117, 135, 167, 204, 212, 224
return statement, 59, 221
rewrite procedure, 98, 117, 135, 168, 204, 211, 225
routine
  addr, 96, 99, 222
  append, 98, 102
  argc, 99, 105, 222
  argv, 99, 105, 222
  arithmetic, 95
  arshft, 97, 108, 171, 222
  asl, 97, 110, 222
  asr, 97, 171, 223
  built-in, 95 to 201
  card, 96, 112, 223
clock, 99, 114, 223
close, 98, 116, 204, 223
concat, 97, 223
date, 99, 118, 223
discard, 121, 223
eof, 135, 204 to 207
eoln, 135, 204 to 209
expo, 96, 123, 223
external option, 92
efirsto, 96, 126, 223
flush, 98, 130, 204, 213, 223
forward option, 91
getenv, 99, 223
getfile, 98, 135, 204, 215, 223
halt, 99, 136, 223
in_range, 96, 138, 223
index, 97, 139, 140, 223
input and output, 203
internal option, 92
land, 97, 142, 223
lastof, 96, 144, 223
length, 97, 146, 223
linelimit, 98, 147, 204, 223
lnot, 97, 149, 223
lor, 97, 150, 224
lshft, 97, 152, 224
ls1, 97, 153
lsr, 97, 153, 224
max, 96, 224
message, 98, 155, 204, 213, 224
min, 96, 157, 224
nonpascal option, 94
null, 99, 157, 224
open, 98, 117, 158, 204, 210, 224
parameters, 85 to 89
private, 84, 92
public, 84
random, 96, 162, 224
read, 98, 163, 204, 205, 207, 209, 211, 224
readln, 98, 135, 163, 204, 207, 209, 211, 224
remove, 98, 166, 204, 224
reset, 98, 117, 135, 167, 204, 212, 224
return statement, 59
rewrite, 98, 117, 135, 169, 204, 211, 225
rshft, 97, 171, 224
seed, 96, 172, 224
seek, 98
sizeof, 96, 176, 224
stlimit, 99, 180, 224
stradd, 97
substr, 97, 183, 224
sysclock, 99, 114, 184, 224
tell, 98, 185, 224
time, 99, 187, 224
trace, 99, 189, 225
trim, 97, 191, 225
type transfer, 99, 193, 225

var declaration, 36, 39
variable option, 92
wallclock, 99, 195, 225
write, 98, 155, 198, 204, 211, 212, 225
writeln, 98, 155, 198, 204, 211, 212, 225
xor, 97, 200, 225
routine parameters, 85 to 89
routines, 79
rshft function, 97, 171, 224

S
-s option to pc command, 2, 3
scope of variables
  private, 80
  public, 80
seed function, 162, 172, 224
seek procedure, 174, 224
set
data types, 38 to 39
  as function return value, 91
  assignment compatibility
    rules, 63
  data representation, 39
  declaring variables, 38
  returning number of elements, 112
  initializing variables, 83
operators, 66, 71
shortreal, 13, 15, 220
signal handler, 214
single, 13, 15, 220, 232
sizeof function, 176, 224
space allocation of data types, 233
special symbols, nonstandard and standard, 2
standard files
  error, 212
  input, 212
  output, 212
statements, 47 to 62
assert, 48, 221
case, 47, 51 to 52, 157, 221
exit, 52 to 53, 221
extensions, 221
for, 52, 56, 221
goto, 48, 54, 59, 221
if, 48, 208, 221
next, 56
repeat, 52, 56, 221
return, 59, 221
while, 52, 56, 206, 209, 221
with, 47, 60, 221
static variable, 14, 18, 21, 23, 28, 37, 39, 83, 222
stderr, 204, 211
stdin, 212
stdout, 211
stlimit procedure, 99, 180, 224
stradd procedure, 182, 224
string
   assignments, 64
   constants, assignments, 65
   data type, 34
   operators, 66, 75
subrange data, 17, 25 to 26
   assignment compatibility rules, 63
   data representation, 25
   declaring variables, 25
   with -xl option, 26
substr function, 97, 183, 224
symbols, 2
sysclock function, 99, 114, 184, 224

T
tab character, 23, 220
tell function, 185, 224
time procedure, 99, 187, 224
trace procedure, 225
trace routine, 189
trim function, 97, 191, 225
type checking of parameters, 88
type declaration, 222
type transfer function, 193, 225
typographical conventions, xvii

U
univ parameter, 85
univ parameter type, 88
univ_ptr, 42, 100, 135
unpacked records
   fixed, 30
   variant, 30
unsigned integer, 17

V
-V0 option to pc command, 3
-V1 option to pc command, 3
value parameter, 86, 88
value parameter, 85
var
declaration, 14, 18, 20, 22, 27, 36, 80 to 83, 222
   attributes, 80
   initialization, 83
   scope, 80
   parameter, 86, 88
var parameter, 85
variable
   attributes, 80
define, 81, 83, 222
derect, 81, 82, 84, 222
global, 80
initialization, 83
local, 80, 83, 100
option, 92, 222
private, 80, 222
public, 80, 81, 83, 222
scope, 80
static, 81, 222
varying data type, 34

W
wallclock function, 99, 195, 225
while statement, 52, 56, 206, 209, 221
with
   alternate form, 60 to 62
   statement, 48, 221
write procedure, 23, 98, 155, 198, 204, 211, 212, 225
writeln procedure, 23, 98, 155, 198, 204, 211, 212, 225

X

-xl option to pc command, 9, 17, 24, 227
  with define attribute, 83
  with nonpascal routine option, 94
xor function, 97, 200, 225