Microsoft® Pascal Reference Manual

Part 1

BEFORE READING THIS MANUAL, REFER TO THE BEGINNING OF THE *MICROSOFT PASCAL USER'S GUIDE* FOR ADDITIONS TO MS-PASCAL.

Manual Part No.
45447-90003

Printed in U.S.A.
1/84

Notice

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy Microsoft Pascal on magnetic tape, disk, or any other medium for any purpose other than the purchaser's personal use.

© Copyright Microsoft Corporation, 1981, 1982, 1983, 1984

If you have comments about the software or these manuals, please complete the Software Problem Report at the back of the Microsoft Pascal Reference Manual and return it to Hewlett Packard.

Microsoft is a registered trademark of Microsoft Corporation.

MS and the Microsoft logo are trademarks of Microsoft Corporation.

Manual Part Number: 45447-90001

Printed in U.S.A. 1/84

Hewlett-Packard
Personal Software Division
3410 Central Expressway
Santa Clara, California 95051

Contents

PART ONE

Introduction

About This Manual Introduction-1
Levels of Microsoft Pascal Introduction-2
Selected Features Introduction-3
Unimplemented Features Introduction-4
Learning More About Pascal Introduction-5

Chapter 1 - Language Overview

Metacommands 1-1
Programs and Compilable Parts of Programs 1-2
Procedures and Functions.................................... 1-5
Statements.. 1-6
Expressions 1-8
Variables 1-9
Constants... 1-10
Types 1-11
Identifiers 1-12
Notation.. 1-13

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

Chapter 2 - Notation

Components of Identifiers 2-1
Separators 2-2
Special Symbols 2-3
Unused Characters.................................... 2-6
Notes on Characters.................................... 2-6

Chapter 3 - Identifiers

What Is an Identifier? 3-1
Declaring an Identifier 3-3
The Scope of Identifiers 3-3
Predeclared Identifiers.................................... 3-4

Chapter 4 - Introduction to Data Types

What Is a Type? 4-1
Declaring Data Types.................................... 4-3
Type Compatibility 4-4

Chapter 5 - Simple Types

Ordinal Types.................................... 5-1
REAL 5-6
INTEGER4.................................... 5-8

Chapter 6 - Arrays, Records, and Sets

Arrays 6-2
Super Arrays 6-3
Records 6-13
Sets 6-18

Chapter 7 - Files

Declaring Files 7-1
The Buffer Variable 7-3
File Structures 7-4
File Access Modes 7-6
The Predeclared Files INPUT and OUTPUT 7-7
Extend Level I/O 7-8
System Level I/O 7-10

Chapter 8 - Reference and Other Types

Reference Types 8-1
PACKED Types 8-9
Procedural and Functional Types 8-10

Chapter 9 - Constants

What Is a Constant? 9-1
Declaring Constant Identifiers 9-2
Numeric Constants 9-3
Character Strings 9-7
Structured Constants 9-8
Constant Expressions 9-10

Chapter 10 - Variables and Values

What Is a Variable? 10-1
Declaring a Variable 10-2
The Value Section 10-3
Using Variables and Values 10-3
Attributes 10-8

Chapter 11 - Expressions

Simple Type Expressions 11-2
Boolean Expressions 11-6
Set Expressions 11-8
Function Designators 11-9
Evaluating Expressions 11-11
Other Features of Expressions 11-13

Chapter 12 - Statements

The Syntax of Pascal Statements 12-1
Simple Statements 12-4
Structured Statements 12-10

PART TWO

Chapter 13 - Introduction to Procedures and Functions

Procedures 13-2
Functions 13-3
Attributes and Directives 13-5
Procedure and Function Parameters 13-14

Chapter 14 - Available Procedures and Functions

Categories of Available Procedures and Functions . . . 14-2
Directory of Functions and Procedures . . . 14-9

Chapter 15 - File-Oriented Procedures and Functions

File System Primitive Procedures and Functions . . . 15-2
Textfile Input and Output . . . 15-11
Extend Level I/O . . . 15-23

Chapter 16 - Compilable Parts of a Program

Programs . . . 16-3
Modules . . . 16-6
Units . . . 16-8

Chapter 17 - Microsoft Pascal Metacommands

Language Level Setting and Optimization . . . 17-3
Debugging and Error Handling . . . 17-4
Source File Control . . . 17-9
Listing File Control . . . 17-12
Listing File Format . . . 17-15
Command Line Switches . . . 17-18

Appendix A - Pascal Syntax Diagrams

Appendix B - Microsoft Pascal Features and the ISO Standard

Microsoft Pascal and the ISO Standard B-2
Summary of Microsoft Pascal Features B-5

Appendix C - Microsoft Pascal and Other Pascals

Implementations of Pascal C-1
Microsoft Pascal and UCSD Pascal C-4

Appendix D - ASCII Character Codes

Appendix E - Summary of Microsoft Pascal Reserved Words

Appendix F - Summary of Available Procedures and Functions

Appendix G - Summary of Microsoft Pascal Metacommands

Appendix H - Messages

Compiler Front End Errors H-2
Compiler Back End Errors H-31
Compiler Internal Errors H-32
Runtime File System Errors H-32
Other Runtime Errors H-37

INTRODUCTION

Microsoft® Pascal, also called MS®-Pascal, is a highly extended, portable version of the Pascal language. Compatible with the International Standards Organization (ISO) proposed standard, its extensions facilitate systems programming as well as applications programming.

You can use MS-Pascal at the ISO standard level for transporting programs to and from other machines. Or, to make full use of the capabilities of a specific computer, you can make your programs more efficient by using the language at its extend or system levels.

The MS-Pascal Compiler generates native machine code; many other Pascal compilers for microcomputers produce intermediate p-code. Programs compiled to native code execute much faster than those compiled to p-code. Thus, with MS-Pascal, you get the programming advantages of a high-level language without sacrificing execution speed. Because of many low-level escapes to the machine level, programs written in MS-Pascal are often comparable in speed to programs written in assembly language.

About This Manual

Chapter 1, "Language Overview," paints a broad picture of MS-Pascal, from the largest elements of the language to the smallest. Later chapters build on this overview, discussing the elements one chapter at a time, starting with the smallest elements of the language and ending with a discussion of programs and other compilable units.

For information on how to use the MS-Pascal Compiler and details on your specific version of MS-Pascal, see the *Microsoft Pascal Compiler User's Guide*.

Levels of Microsoft Pascal

MS-Pascal is organized into three levels: standard, extend, and system. The features of each level are discussed in more detail in Appendix B, "Microsoft Pascal Features and the ISO Standard." Briefly, the differences among the three levels are as follows:

1. Standard level

   At the standard level, programs must conform to the ISO standard. Programs you create at this level are portable to and from machines running other ISO-compatible Pascal compilers. There are some minor MS-Pascal extensions to the standard that won't be caught as errors at this level. For details of these extensions, as well as other issues regarding the ISO standard, see Appendix B, "Microsoft Pascal Features and the ISO Standard." In this manual, the phrases "standard Pascal," "the ISO standard," and "at the standard level of MS-Pascal" are generally synonymous.

2. Extend level

   The extend level is intended for structured and relatively safe extensions to the ISO standard. Programs you create at this level are portable among all machines that run MS-Pascal.

3. System level

   The system level includes all features available at the extend level. It also includes some unstructured, machine-oriented extensions, such as address types and ability to access all file control block fields, which are useful for systems programming.

In this manual, extensions to standard Pascal are called "features." A complete list of these features and the level at which they are available is given in Appendix B, "Microsoft Pascal Features and the ISO Standard." Selected features are described briefly in the following list.

In addition to these three levels, MS-Pascal has a large number of "metacommands," that is, directives with which you can control the compiler. See Chapter 17, "Microsoft Pascal Metacommands," for more information.

Selected Features

The following list includes just some of the features available at the extend and system levels of MS-Pascal. For a complete list, see Appendix B, "Summary of Microsoft Pascal Features."

1. Underscore in identifiers, which improves readability.
2. Nondecimal numbering (hexadecimal, octal, and binary), which facilitates programming at the byte and bit level.
3. Structured constants, which you may declare in the declaration section of a program or use in statements.
4. Variable length strings (type LSTRING), as well as special predeclared procedures and functions for LSTRINGs, which overcome standard Pascal's poor string handling capabilities.
5. Super arrays, a special variable length array whose declaration permits passing arrays of different lengths to a reference parameter, as well as dynamic allocation of arrays of different lengths.
6. Predeclared unsigned BYTE (0-255) and WORD (0-65535) types, which facilitate programming at the system level.
7. Address types (segmented and unsegmented), which allow manipulation of actual machine addresses at the system level.
8. String reads, which allow the standard procedures READ and READLN to read strings as structures rather than character by character.
9. Interface to assembly language, provided by PUBLIC and EXTERN procedures, functions, and variables, which allows low-level interfacing to assembly language and library routines.
10. VALUE section, where you may declare the initial constant values of variables in a program.
11. Function return values of a structured type as well as of a simple type.
12. Direct (random access) files, accessible with the SEEK procedure, which enhance standard Pascal's file accessing capabilities.
13. Lazy evaluation, a special internal mechanism for interactive files that allows normal interactive input from terminals.

14. Structured BREAK and CYCLE statements, which allow structured exits from a FOR, REPEAT, or WHILE loop; and the RETURN statement, which allows a structured exit from a procedure or function.

15. OTHERWISE in CASE statements, whereby you avoid explicitly specifying each CASE constant. OTHERWISE is also permitted with variant records.

16. STATIC attribute for variables, which allows you to indicate that a variable is to be allocated at a fixed location in memory rather than on the stack.

17. ORIGIN attribute, which may be given to variables, procedures, and functions to indicate their absolute location in memory.

18. INTERRUPT attribute for procedures, which signals the compiler to give the procedure a special calling sequence that saves the machine status on the stack upon entry and restores the machine status upon exit.

19. Separate compilation of portions of a program (units and modules).

20. Conditional compilation, using conditional metacommands in your MS-Pascal source file to switch on or off compilation of parts of the source.

Unimplemented Features

The following features either are not presently implemented or are implemented only as described in the following list:

1. OTHERWISE is not accepted in RECORD declarations.
2. Code is generated for PURE functions, but no checking is done.
3. The extend level operators SHL, SHR, and ISR are not available.
4. ENABIN, DISBIN, and VECTIN library routines are not available. The INTERRUPT attribute is ignored.
5. No checking is done for invalid GOTOs and uninitialized REAL values.
6. READ, READLN, and DECODE cannot have M and N parameters.

7. Enumerated I/O, for reading and writing enumerated constants as strings, is not available.
8. The metacommands $TAGCK, $STANDARD, $EXTEND, and $SYSTEM can be given, but have no effect.
9. The $INCONST metacommand does not accept string constants.

Learning More About Pascal

The manuals in this package provide complete reference information for your implementation of the MS-Pascal Compiler. They do not, however, teach you how to write programs in Pascal. If you are new to Pascal or need help in learning to program, we suggest you read any of the following books:

Findlay, W., and Watt, D. F. *Pascal: An Introduction to Methodical Programming*. Pittman: London, 1978.

Holt, Richard C., and Hume, J. N. P. *Programming Standard Pascal.* Reston Publishing Company: Reston, Va., 1980.

Jensen, Kathleen, and Wirth, Niklaus. *Pascal User Manual and Report.* Springer-Verlag: New York, 1974, 1978.

Koffman, E. B. *Problem Solving and Structured Programming in Pascal.* Addison-Wesley Publishing Company: Reading, Mass., 1981.

Schneider, G. M., Weinhart, S. W., and Perlman, D. M. *An Introduction to Programming and Problem Solving With Pascal.* John Wiley & Sons: New York, second edition, 1982.

Chapter 1

LANGUAGE OVERVIEW

In this chapter you will find a summary description of Microsoft Pascal from the largest elements of the language to the smallest. Each of the remaining chapters of the manual discusses these elements in detail, from the smallest element (notation) to the largest (metacommands).

Metacommands

The MS-Pascal metacommands provide a control language for the MS-Pascal Compiler. The metacommands let you specify options that affect the overall operation of a compilation. For example, you can conditionally compile different source files, generate a listing file, or enable or disable runtime error checking code.

Metacommands are inserted inside comment statements. All of the metacommands begin with a dollar sign ($). Some may also be given as switches when you invoke the compiler.

Although most implementations of Pascal have some type of compiler control, the MS-Pascal metacommands are not part of standard Pascal and hence are not portable.

These are the metacommands currently available:

```
$BRAVE                      $PAGEIF
$DEBUG                      $PAGESIZE
$ENTRY                      $POP
$ERRORS                     $PUSH
$EXTEND                     $RANGECK
$GOTO                       $REAL
$INCLUDE                    $ROM
$INCONST                    $RUNTIME
$INDEXCK                    $SIMPLE
$INTICK                     $SIZE
$IF $THEN $ELSE $END        $SKIP
$INTEGER                    $SPEED
$LINE                       $STACKCK
$LINESIZE                   $STANDARD
$LIST                       $SUBTITLE
$MATHCK                     $SYMTAB
$MESSAGE                    $SYSTEM
$NILCK                      $TAGCK
$OCODE                      $TITLE
$OPTBUG                     $WARN
$PAGE
```

See Chapter 17, "Microsoft Pascal Metacommands," for a complete discussion of metacommands.

Programs and Compilable Parts of Programs

The MS-Pascal Compiler processes programs, modules, and implementations of units. Collectively, these compilable programs and parts of programs are referred to as compilands. You can compile modules and implementations of units separately and then later link them to a program without having to recompile the module or unit.

The fundamental unit of compilation is a program. A program has three parts:

1. The program heading identifies the program and gives a list of program parameters.
2. The declaration section follows the program heading and contains declarations of labels, constants, types, variables, functions, and procedures. Some of these declarations are optional.

3. The body follows all declarations. It is enclosed by the reserved words BEGIN and END and is terminated by a period. The period is the signal to the compiler that it has reached the end of the source file.

The following program illustrates this three-part structure:

```
{Program heading}
PROGRAM FRIDAY (INPUT,OUTPUT);

{Declaration section}
LABEL 1;
CONST DAYS_IN_WEEK = 7;
TYPE  KEYBOARD_INPUT = CHAR;
VAR   KEYIN: KEYBOARD_INPUT;

{Program body}
BEGIN
   WRITE('IS TODAY FRIDAY? ');
1: READLN(KEYIN);
   CASE KEYIN OF
   'Y', 'y' : WRITELN('It''s Friday.');
   'N', 'n' : WRITELN('It''s not Friday.');
   OTHERWISE
      WRITELN('Enter Y or N.');
      WRITE('Please re-enter: ');
      GOTO 1
   END
END.
```

This three-part structure (heading, declaration section, body) is used throughout the Pascal language. Procedures, functions, modules, and units are all similar in structure to a program.

Modules are program-like units of compilation that contain the declaration of variables, constants, types, procedures, and functions, but no program statements. You can compile a module separately and later link it to a program, but it cannot be executed by itself.

Example of a module:

```
{Module heading}
MODULE MODPART;

{Declaration section}
CONST PI = 3.14;

PROCEDURE PARTA;
   BEGIN
     WRITELN ('parta')
   END;

{Body}
END.
```

A module, like a program, ends with a period. Unlike a program, a module contains no program statements.

A unit has two sections: an interface and an implementation. Like a module, an implementation may be compiled separately and later linked to the rest of the program. The interface contains the information that lets you connect a unit to other units, modules, and programs.

Example of a unit:

```
{Heading for interface}
INTERFACE;
UNIT MUSIC (SING, TOP);

{Declarations for interface}
VAR TOP : INTEGER;
PROCEDURE SING;

{Body of interface}
BEGIN
END;

{Heading for implementation}
IMPLEMENTATION OF MUSIC;

{Declaration for implementation}
PROCEDURE SING;
BEGIN
   FOR I := 1 TO TOP DO
   BEGIN
     WRITE ('FA '); WRITELN ('LA LA')
   END
END;

{Body of implementation}
BEGIN
   TOP := 5
END.
```

A unit, like a program or a module, ends with a period.

Modules and units let you develop large structured programs that can be broken into parts. This practice is advantageous in the following situations:

1. If a program is large, breaking it into parts makes it easier to develop, test, and maintain.
2. If a program is large and recompiling the entire source file is time consuming, breaking the program into parts saves compilation time.
3. If you intend to include certain routines in a number of different programs, you can create a single object file that contains these routines and then link it to each of the programs in which the routines are used.
4. If certain routines have different implementations, you might place them in a module to test the validity of an algorithm and later create and implement similar routines in assembly language to increase the speed of the algorithm.

See Chapter 16, "Compilable Parts of a Program," for a complete discussion of programs, modules, and units.

Procedures and Functions

Procedures and functions act as subprograms that execute under the supervision of a main program. However, unlike programs, procedures and functions can be nested within each other and can even call themselves. Furthermore, they have sophisticated parameter-passing capabilities that programs lack.

Procedures are invoked as statements; functions can be invoked in expressions wherever values are called for.

A procedure declaration, like a program, has a heading, a declaration section, and a body.

Example of a procedure declaration:

```
{Heading}
PROCEDURE COUNT_TO(NUM : INTEGER);

{Declaration section}
VAR I : INTEGER;

{Body}
BEGIN
  FOR I := 1 TO NUM DO WRITELN (I)
END;
```

A function is a procedure that returns a value of a particular type; hence, a function declaration must indicate the type of the return value.

Example of a function declaration:

```
{Heading}
FUNCTION ADD (VAL1, VAL2 : INTEGER): INTEGER;

{Declaration section empty}

{Body}
BEGIN
   ADD := VAL1 + VAL2
END;
```

Procedures and functions look somewhat different from programs, in that their parameters have types and other options. Like the body of a program, the body of a procedure or a function is enclosed by the reserved words BEGIN and END; however, a semicolon rather than a period follows the word "END".

Declaring a procedure or function is entirely distinct from using it in a program. For example, the procedure and function declared above might actually appear in a program as follows:

```
TARGET_NUMBER := ADD (5, 6);
COUNT_TO (TARGET_NUMBER);
```

See Chapter 13, "Introduction to Procedures and Functions," for a complete discussion of procedures and functions.

See Chapter 14, "Available Procedures and Functions," and Chapter 15, "File-Oriented Procedures and Functions," for a discussion of procedures and functions that are predeclared as part of the MS-Pascal language.

Statements

Statements perform actions, such as computing, assigning, altering the flow of control, and reading and writing files. Statements are found in the bodies of programs, procedures, and functions and are executed as a program runs. MS-Pascal statements perform the actions shown in Table 1.1.

Table 1.1 Summary of Microsoft Pascal Statements

Statement	Purpose
Assignment	Replaces the current value of a variable with a new value.
BREAK	Exits the currently executing loop.
CASE	Allows for the selection of one action from a choice of many, based on the value of an expression.
CYCLE	Starts the next iteration of a loop.
FOR	Executes a statement repeatedly while a progression of values is assigned to a control variable.
GOTO	Continues processing at another part of the program.
IF	Together with THEN and ELSE, allows for conditional execution of a statement.
Procedure call	Invokes a procedure with actual parameter values.
REPEAT	Repeats a sequence of statements one or more times, until a Boolean expression becomes true.
RETURN	Exits the current procedure, function, program, or implementation.
WHILE	Repeats a statement zero or more times, until a Boolean expression becomes false.
WITH	Opens the scope of a statement to include the fields of one or more records, so that you can refer to the fields directly.

See Chapter 12, "Statements," for a detailed discussion of each of these statements.

Expressions

An expression is a formula for computing a value. It consists of a sequence of operators (which indicate the action to be performed) and operands (the value on which the operation is performed). Operands may contain function invocations, variables, constants, or even other expressions. In the following expression, plus (+) is an operator, while A and B are operands:

```
A + B
```

There are three basic kinds of expressions:

1. Arithmetic expressions perform arithmetic operations on the operands in the expression.
2. Boolean expressions perform logical and comparison operations with Boolean results.
3. Set expressions perform combining and comparison operations on sets, with Boolean or set results.

Expressions always return values of a specific type. For instance, if A, B, C, and D are all REAL variables, then the following expression evaluates to a REAL result:

```
A * B + (C / D) + 12.3
```

Expressions may also include function designators:

```
ADDREAL (2, 3) + (C / D)
```

ADDREAL is a function that has been previously declared in a program. It has two REAL value parameters, which it adds together to obtain a total. This total is the return value of the function, which is then added to (C / D).

Expressions are not statements, but may be components of statements. In the following example, the entire line is a statement; only the portion after the equal sign is an expression:

```
X := 2 / 3 + A * B
```

See Chapter 11, "Expressions," for a detailed discussion of expressions.

Variables

A variable is a value that is expected to change during the course of a program. Every variable must be of a specific data type.

After you declare a variable in the heading or declaration section of a compiland, procedure, or function, it may be used in any of the following ways:

1. You may initialize it, in the VALUE section of a program.
2. You may assign it a value, with an assignment statement.
3. You may pass it as a parameter to a procedure or function.
4. You may use it in an expression.

The VALUE section is an MS-Pascal feature that applies only to statically allocated variables (variables with a fixed address in memory). First you declare the variables, as shown in the following example:

```
VAR I, J, K, L : INTEGER;
```

Then you assign them initial values in the VALUE section:

```
VALUE I := 1; J := 2; K := 3; L := 4;
```

Later, in statements, the variables can be assigned to, and used as operands in expressions:

```
I := J + K + L;
J := 1 + 2 + 3;
K := (J * K) + 9 + (L DIV J);
```

See Chapter 10, "Variables and Values," for a complete discussion of variables.

Constants

A constant is a value that is not expected to change during the course of a program. At the standard level, a constant may be:

1. a number, such as 1.234 and 100
2. a string enclosed in single quotation marks, such as 'Miracle' or 'A1207'
3. a constant identifier that is a synonym for a numeric or string constant

You declare constant identifiers in the CONST section of a compiland, procedure, or function:

```
CONST  REAL_CONST = 1.234;
       MAX_VAL    = 100;
       TITLE      = 'Pascal';
```

Because the order of declarations is flexible in MS-Pascal, you can declare constants anywhere in the declaration section of a compilable part of a program, any number of times.

Constants are closely tied to the concepts of variables and types. Variables are all of some type; types, in turn, designate a range of assumable values. These values, ultimately, are all constants.

Two powerful extensions in MS-Pascal are structured constants and constant expressions.

1. VECTOR, in the following example, is an array constant:

   ```
   CONST VECTOR = VECTORTYPE (1,2,3,4,5);
   ```

2. MAXVAL, in the following example, is a constant expression (A, B, C, and D must also be constants):

   ```
   CONST MAXVAL = A * (B DIV C) + D - 5;
   ```

See Chapter 9, "Constants," for a complete discussion of these and other aspects of constants.

Types

Much of Pascal's power and flexibility lies in its data typing capability. Although a great variety of data types are available, they may be divided into three broad categories: simple, structured, and reference types.

1. A simple data type represents a single value; a structured type represents a collection of values. The simple types include the following:

INTEGER	enumerated
WORD	subrange
CHAR	REAL
BOOLEAN	INTEGER4

2. The structured types include the following:

   ARRAY
   RECORD
   SET
   FILE

3. Reference types allow recursive definition of types in an extremely powerful manner.

All variables in Pascal must be assigned a data type. A type is either predeclared (e.g., INTEGER and REAL) or defined in the declaration section of a program. The following sample type declaration creates a type that can store information about a student:

```
TYPE
   STUDENT = RECORD
      AGE       : 5..18;
      SEX       : (MALE, FEMALE);
      GRADE     : INTEGER;
      GRADE_PT  : REAL;
      SCHEDULE  : ARRAY [1..10] OF CLASSES
END;
```

For a detailed discussion of data types, see the following chapters:

Chapter 4, "Introduction to Data Types"
Chapter 5, "Simple Types"
Chapter 6, "Arrays, Records, and Sets"
Chapter 7, "Files"
Chapter 8, "Reference and Other Types"

Identifiers

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program. Procedures and functions must have identifiers; constants, type, and variables may have identifiers (and it is useful if they do).

You, the programmer, make up most of the identifiers in a program and assign them meaning in declarations. Other identifiers are the names of variables, data types, procedures, and functions that are built into the language and need not be declared.

An identifier must begin with a letter (A through Z and a through z). The initial letter may be followed by any number of letters, digits (0 through 9), or underscore characters. The compiler ignores the case of letters; thus, "A" and "a" are equivalent.

The underscore in MS-Pascal is significant. Thus, the following are not identical:

```
FOREST
FOR_EST
```

The only restriction on identifiers is that you must not choose a Pascal reserved word (see Chapter 2, "Reserved Words," for a discussion of reserved words; see Appendix E, "Summary of Microsoft Pascal Reserved Words," for a complete list).

Furthermore, most compilers have some restriction either on the absolute length of an identifier or on the number of characters that are considered significant. See Appendix A, "Version Specifics," in your *Microsoft Pascal Compiler User's Guide* for any limitations imposed by your version of the compiler.

See Chapter 3, "Identifiers," for a complete discussion of identifiers in MS-Pascal.

Notation

The basis of all Pascal programs is an irreducible set of symbols with which the higher syntactic components of the language are created.

The underlying notation is the ASCII character set, divided into the following syntactic groups:

1. Identifiers are the names given to individual instances of components of the language.
2. Separators are characters that delimit adjacent numbers, reserved words, and identifiers.
3. Special symbols include punctuation, operators, and reserved words.
4. Some characters are unused by MS-Pascal but are available for use in a comment or string literal.

A good understanding of this notation will increase your productivity by reducing the number of subtle syntactic errors in a program. See Chapter 2, "Notation," for a detailed discussion of MS-Pascal notation.

Chapter 2

NOTATION

All components of the MS-Pascal language are constructed from the standard ASCII character set. Characters make up lines, each of which is separated by a character specific to your operating system. Lines make up files.

Within a line, individual characters or groups of characters fall into one (or more) of four broad categories:

1. components of identifiers
2. separators
3. special symbols
4. unused characters

Components of Identifiers

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program.

The use of identifiers is described thoroughly in Chapter 3, "Identifiers." This section discusses only how to construct them.

Identifiers must begin with a letter; subsequent components may include letters, digits, and underscore characters.

Although, in theory, there is no limit on the number of characters in an identifier, most implementations restrict the length in some way. See Appendix A, "Version Specifics," in your *Microsoft Pascal Compiler User's Guide* for any limitations that may apply to your machine.

Letters

In identifiers, only the uppercase letters A through Z are significant. You may use lowercase letters for identifiers in a source program. However, the MS-Pascal Compiler converts all lowercase letters in identifiers to the corresponding uppercase letters.

Letters in comments or in string literals may be either uppercase or lowercase; the difference is significant. No mapping of lowercase to uppercase occurs in either comments or string literals.

Digits

Digits in Pascal are the numbers zero through nine. Digits can occur in identifiers (for example, AS129M) or in numeric constants (for example, 1.23 and 456).

The Underscore Character

The underscore (_) is the only nonalphanumeric character allowed in identifiers. The underscore is significant in MS-Pascal. Use it like a space to improve readability.

For example, the identifiers in the right-hand column below are easier to read than those in the left-hand column:

```
POWEROFTEN        POWER_OF_TEN
MYDOGMAUDE        MY_DOG_MAUDE
```

Separators

Separators delimit adjacent numbers, reserved words, and identifiers, none of which should have separators embedded within them.

A separator can be any of the following:

1. the space character
2. the tab character
3. the form feed character
4. the new line marker
5. the comment

Comments in standard Pascal take one of these forms:

```
{This is a comment, enclosed in braces.}
(*This is an alternate form of comment.*)
```

The second form is generally used if braces are unavailable on a particular machine. Comments in either of these forms may span more than one line.

At the extend level, MS-Pascal also allows comments that begin with an exclamation point:

```
! The rest of this line is a comment.
```

For comments in this form, the new line character delimits the comment.

Nested comments are permitted in MS-Pascal, so long as each level has different delimiters. Thus, when a comment is started, the compiler ignores succeeding text until it finds the matching end-of-comment. However, such nesting may not be portable.

Always use separators between identifiers and numbers. If you fail to do so, the compiler will generally issue an error or warning message. In a few cases, the MS-Pascal Compiler accepts a missing separator without generating an error message.

For example, at extend level,

```
100MOD#127
```

is accepted as 100 MOD #127, where #127 is a hexadecimal number. However,

```
100MOD127
```

is assumed to be 100 followed by the identifier MOD127.

Special Symbols

Special symbols fall into three categories:

1. punctuation
2. operators
3. reserved words

Punctuation

Punctuation in MS-Pascal serves a variety of purposes, including those shown in Table 2.1.

Table 2.1. Summary of Punctuation in Microsoft Pascal

Symbol	Purpose
{ }	Braces delimit comments.
[ ]	Brackets delimit array indices, sets, and attributes. They may also replace the reserved words BEGIN and END in a program.
( )	Parentheses delimit expressions, parameter lists, and program parameters.
'	Single quotation marks enclose string literals.
:=	The colon-equals symbol assigns values to variables in assignment statements and in VALUE sections.
;	The semicolon separates statements and declarations.
:	The colon separates variables from types and labels from statements.
=	The equal sign separates identifiers and type clauses in a TYPE section.
,	The comma separates the components of lists.
..	The double period denotes a subrange.
.	The period designates the end of a program, indicates the fractional part of a real number, and also delimits fields in a record.
^	The up arrow denotes the value pointed to by a reference value.
#	The number sign denotes nondecimal numbers.
$	The dollar sign prefixes metacommands.

Operators

Operators are a form of punctuation that indicate some operation to be performed. Some are alphabetic, others are one or two nonalphanumeric characters. Any operators that consist of more than one character must not have a separator between characters.

The operators that consist only of nonalphabetic characters are the following:

```
+ - * / > < = < > < = >=
```

Some operators (e.g., NOT and DIV) are reserved words instead of nonalphabetic characters.

See Chapter 11, "Expressions," for a complete list of the nonalphabetic operators and a discussion of the use of operators in expressions.

Reserved Words

Reserved words are a fixed part of the MS-Pascal language. They include, for example, statement names (e.g., BREAK) and words like BEGIN and END that bracket the main body of a program. See Appendix E, "Summary of Microsoft Pascal Reserved Words," for a complete list.

You cannot create an identifier that is the same as any reserved word. You may, however, declare an identifier that contains within it the letters of a reserved word (for example, the identifier DOT containing the reserved word DO).

There are several categories of reserved words in MS-Pascal:

1. reserved words for standard level MS-Pascal
2. reserved words added for extend level MS-Pascal features
3. reserved words added for system level MS-Pascal features
4. names of attributes
5. names of directives

See Appendix E, "Summary of Microsoft Pascal Reserved Words," for a complete list of reserved words. Look in the index for the pages where each is discussed in this manual.

Unused Characters

A few printing characters are not used in MS-Pascal:

% & " | ~ '

You may, however, use them within comments or string literals.

A number of other nonprinting ASCII characters will generate error messages if you use them in a source file other than in a comment or string literal:

1. the characters from CHR (0) to CHR (31), except the tab and form feed, CHR (9) and CHR (12), respectively
2. the characters from CHR (127) to CHR (255)

The tab character, CHR (9), is treated like a space and is passed on to the listing file. A form feed, CHR (12), is treated like a space and starts a new page in the listing file.

Notes on Characters

This sections discusses special notational properties of the MS-Pascal language not mentioned elsewhere in this chapter.

Characters within a comment or string literal are always legal and have no special effects.

MS-Pascal allows the following substitutions:

If Your Keyboard Lacks	Use This Instead
[	].
]	.]
^	@ or ?
@	^ or ?

The substitution of a question mark (?) for an up arrow (^) is a minor extension to the ISO standard.

Table 2.2 gives a list of pairs of printing characters that are the same ASCII character.

Table 2.2 Equivalent ASCII Characters

ASCII	Prints as	Equivalent Characters
CHR (94)	^	up arrow, caret
CHR (95)	_	underscore, left arrow
CHR (35)	#	number sign, English pound sign
CHR (36)	$	dollar sign, scarab (circle with four spikes)

Chapter 3

IDENTIFIERS

What Is an Identifier?

Identifiers are names that denote the constants, variables, data types, procedures, functions, and other elements of a Pascal program. Procedures and functions must have identifiers; constants, types, and variables may have identifiers (and it is useful if they do).

Some identifiers are predeclared; others you declare in a declaration section. Standard Pascal allows identifiers for the following elements of the Pascal language:

- types
- constants
- variables
- procedures
- functions
- programs
- fields and tag fields in records

The following MS-Pascal features at the extend level also require identifiers:

- super array types
- modules
- units
- statement labels

An identifier consists of a sequence of alphanumeric characters or underscore characters. The first character must be alphabetic. Underscores in identifiers are allowed, and are significant, at all levels of MS-Pascal. Two underscores in a row or an underscore at the end of an identifier are permitted.

Subject to the restrictions noted below, identifiers can be as long as you want. They must, however, fit on a single line. At least the first 19 characters of an identifier are significant; in some versions, as many as 31 characters are significant.

An identifier longer than the significance length generates a warning but not an error message; the excess characters are ignored by the compiler. See Appendix A, "Version Specifics," in your *Microsoft Pascal Compiler User's Guide* for the significance length in your implementation.

Standard Pascal allows unsigned integers as statement labels. Statement labels have the same scope rules as identifiers (see "The Scope of Identifiers" below). Leading zeros are not significant.

Extend level MS-Pascal allows labels that are normal alphabetic identifiers.

The identifiers used for a program, module, or unit, as well as identifiers with the PUBLIC or EXTERN attribute, are passed to the linker. The operating system of the machine on which you plan to link and run a compiled MS-Pascal program may impose further identifier length restrictions on identifiers used as linker global symbols. Furthermore, the object code listing and debugger symbol table may truncate variable and procedural identifiers to six characters.

Writing programs for use with other compilers and operating systems imposes an additional constraint on a program. Such a program must conform to the identifier restrictions for the worst possible case.

For portability in general, the following practices are recommended:

1. Make all identifiers unique in their first eight characters.
2. Make external identifiers unique in their first six characters.
3. Limit statement labels to four digits without leading zeros.

Identifiers of seven characters or fewer save space during compilation.

Note

All identifiers used internally by the runtime system are four alphabetic characters followed by the characters QQ. Avoid this form when creating new names yourself.

Declaring an Identifier

You declare identifiers and associate them with language objects in the declaration section of a program, module, interface, implementation, procedure, or function. Examples of identifiers, the objects they represent, and the syntax used to declare them are shown below in Table 3.1. Although the details vary, the basic form of the declaration of the identifier for each of these elements is similar.

Table 3.1. Declaring Identifiers

Object	Identifier	Declaration
Program	Z	PROGRAM Z (INPUT, OUTPUT)
Module	XXX	MODULE XXX
Interface	UUU	INTERFACE; UNIT UUU
Implementation	UUU	IMPLEMENTATION of UUU
Constant	DAYS	CONST DAYS = 365
Type	LETTERS	TYPE LETTERS = 'A'..'Z'
Record fields	X, Y, Z	TYPE A = RECORD X, Y ,Z : REAL END
Variable	J	VAR J : INTEGER
Label	1	LABEL 1
Label	HAWAII	LABEL HAWAII
Procedure	BANG	PROCEDURE BANG
Function	FOO	FUNCTION FOO: INTEGER

The Scope of Identifiers

An identifier is defined for the duration of the procedure, function, program, module, implementation, or interface in which you declare it. This holds true for any nested procedures or functions. An identifier's association must be unique within its scope; that is, it must not name more than one thing at a time.

A nested procedure or function can redefine an identifier only if the identifier has not already been used in it. However, the compiler does not identify such redefinition as an error, but will generally use the first definition until the second occurs. A special exception for reference types is discussed in Chapter 8, "Notes on Reference Types."

Predeclared Identifiers

A number of identifiers are already a part of the MS-Pascal language. This category includes the identifiers for predeclared types, super array types, constants, file variables, functions, and procedures.You can use them freely, without declaring them. However, they differ from reserved words in that you may redefine them whenever you wish.

At the standard level, the following identifiers are predeclared:

```
ABS        FALSE      OUTPUT     ROUND
ARCTAN     FLOAT      PACK       SIN
BOOLEAN    GET        PAGE       SQR
CHAR       INPUT      PRED       SQRT
CHR        INTEGER    PUT        SUCC
COS        LN         READ       TEXT
DISPOSE    MAXINT     READLN     TRUE
EOF        NEW        REAL       TRUNC
EOLN       ODD        RESET      UNPACK
EXP        ORD        REWRITE    WRITE
                                 WRITELN
```

Features at the extend and system levels add the following to the list of predeclared identifiers in MS-Pascal.

1. String intrinsics

```
CONCAT       INSERT
COPYLST      POSITN
COPYSTR      SCANEQ
DELETE       SCANNE
```

2. Extend level intrinsics

```
ABORT      HIBYTE
BYWORD     LOBYTE
DECODE     LOWER
ENCODE     RESULT
EVAL       SIZEOF
           UPPER
```

3. System level intrinsics

```
FILLC      MOVESL
FILLSC     MOVESR
MOVEL      RETYPE
MOVER
```

4. Extend level I/O

```
ASSIGN      READFN
CLOSE       READSET
DIRECT      SEEK
DISCARD     SEQUENTIAL
FCBFQQ      TERMINAL
FILEMODES
```

5. INTEGER4 type

```
BYLONG       LOWORD
FLOATLONG    MAXINT4
HIWORD       ROUNDLONG
INTEGER4     TRUNCLONG
```

6. Super array type

```
LSTRING
NULL
STRING
```

7. WORD type

```
MAXWORD
WORD
WRD
```

8. Miscellaneous

```
ADRMEM       INTEGER2
ADSMEM       REAL4
BYTE         REAL8
INTEGER1     SINT
```

Chapter 4

INTRODUCTION TO DATA TYPES

What Is a Type?

A type is the set of values that a variable or value can have within a program. Types are either predeclared or declared explicitly.

For example, the types INTEGER and REAL are predeclared, while the type ARRAY [1..10] OF INTEGER is declared explicitly. An explicitly declared type may also be given a type identifier. To accomplish this latter task, a type declaration is required.

Types in MS-Pascal fall into three broad categories: simple, structured, and reference types. Table 4.1 gives a breakdown of the types in each of these groups. The remainder of this chapter discusses types in general; Chapters 5 through 8 discuss the different groups in detail.

Table 4.1. Categories of Types in Microsoft Pascal

Category	Includes	Comments / Examples
Simple Types	Ordinal types	
	INTEGER	-MAXINT..MAXINT
	WORD	0..MAXWORD
	CHAR	CHR(0)..CHR(255)
	BOOLEAN	(FALSE,TRUE)
	enumerated types	e.g., (RED,BLUE)
	subrange types	e.g., 100..5000
	REAL4, REAL8	
	INTEGER4	-MAXINT4..MAXINT4
Structured Types	ARRAY OF type	
	general (OF any type)	
	SUPER ARRAY (OF type)	
	STRING (n)	[1..n] of CHAR
	LSTRING (n)	[0..n] of CHAR
	RECORD •	
	SET OF type	
	FILE OF	
	general (binary) files	
	TEXT	Like FILE OF CHAR
Reference Types	Pointer Types	e.g., ^TREETIP
	ADR OF type	Relative address
	ADS OF type	Segmented address
Procedural and Functional Types		Only as parameter type

Declaring Data Types

The type declaration associates an identifier with a type of value. You declare types in the TYPE section of a program, procedure, function, module, interface, or implementation (not in the heading of a procedure or function).

A type declaration consists of an identifier followed by an equal sign and a type clause.

Examples of type definitions:

```
TYPE LINE = STRING (80);
     PAGE = RECORD
         PAGENUM : 1..499;
         LINES : ARRAY [1..60] OF LINE;
         FACE : (LEFT, RIGHT);
         NEXTPAGE : ^PAGE;
         END;
```

After declaring the data types, you declare variables of the types just defined in the VAR section of a program, procedure, function, module, or interface, or in the heading of a procedure or function. The following sample VAR section declares variables of the types in the preceding sample TYPE section:

```
VAR PARAGRAPH : LINE;
    BOOK : PAGE;
```

Because a type identifier is not defined until its declaration is processed by the compiler, a recursive type declaration such as the following is illegal:

```
T = ARRAY [0..9] OF T;
```

Reference types require a standard exception to this rule and are discussed in Chapter 8, "Reference and Other Types."

A special feature of MS-Pascal is a category called super types. A super type declaration determines the set of types that designators of that super type can assume; it also associates an identifier with the super type. Super type declarations also occur in the TYPE section. The only super types currently available in MS-Pascal are super arrays.

Type Compatibility

MS-Pascal follows the ISO standard for type compatibility, with some additional rules added for super array types, LSTRINGs, and constant coercions (i.e., forced changes in the type of a constant). Type transfer functions, to override the typing rules, are available with some MS-Pascal features.

Two types can be "identical," "compatible," or "incompatible." An expression may or may not be "assignment compatible" with a variable, value parameter, or array index.

Type Identity and Reference Parameters

Two types are identical if they have the identical identifier or if the identifiers are declared equivalent with a type definition like the following:

```
TYPE T1 = T2;
```

"Identical" types are truly identical in MS-Pascal: there is no difference between types T1 and T2 in the example above. Type identity is based on the name of the types, and not on the way they are declared or structured. Thus, for example, T1 and T2 are not identical in the following declarations:

```
TYPE T1 = ARRAY [1..10] OF CHAR;
     T2 = ARRAY [1..10] OF CHAR;
```

Actual and formal reference parameters must be of identical types. Or, if a formal reference parameter is of a super array type, the actual parameter must be of the same super array type or a type derived from it. Two record or array types must be identical for assignment.

The only exception is for strings. Here, actual parameters of type CHAR, STRING, STRING (n), LSTRING,and LSTRING (n) are compatible with a formal parameter of super array type STRING. Also, the type of a string constant will change to any LSTRING type with a large enough bound. For example, the type of 'ABC' will change to LSTRING (5) if necessary.

Furthermore, an actual parameter of any FILE type may be passed to a formal parameter of a special record type FCBFQQ. Similarly, an actual parameter of type FCBFQQ may be passed to a formal parameter of any file type. See Chapter 7, "System Level I/O," for a description of the FCBFQQ type.

STRING (n) is a shorthand notation for:

```
PACKED ARRAY [1..n] OF CHAR
```

The two types are identical. However, because variables with the type LSTRING are treated specially in assignments, comparisons, READs, and WRITEs, LSTRING (n) is not a shorthand notation for PACKED ARRAY [0..n] OF CHAR. The two types are not identical, compatible, or assignment compatible. See Chapter 6, "Using STRINGs and LSTRINGs," for further information on string types.

Type Compatibility and Expressions

Two simple or reference types are compatible if any of the following is true:

1. They are identical.
2. They are both ADR types.
3. They are both ADS types.
4. One is a subrange of the other.
5. They are subranges of compatible types.

Two structured types are compatible if any of the following is true:

1. They are identical.
2. They are SET types with compatible base types.
3. They are STRING derived types of equal length.
4. They are LSTRING derived types.

However, two structured types are incompatible if any of the following is true:

1. Either type is a FILE or contains a FILE.
2. Either type is a super array type.
3. One type is PACKED and the other is not.

Two values must be of compatible types when combined with an operator in an expression. (Most operators have additional limitations on the type of their operands. See Chapter 11, "Expressions," for details.) A CASE index expression type must be compatible with all CASE constant values. Note that two sets are never compatible if one is PACKED and the other is not.

Assignment Compatibility

Some types are implicitly compatible. This permits assignment across type boundaries. For instance, assume you declare the following variables:

```
VAR DESTINATION : T_DEST;
    SOURCE : T_SOURCE;
```

SOURCE is assignment compatible with DESTINATION (i.e., DESTINATION := SOURCE is permitted) if one of the following is true:

1. T_SOURCE and T_DEST are identical types.
2. T_SOURCE and T_DEST are compatible and SOURCE has a value in the range of subrange type T_DEST.
3. T_DEST is of type REAL and T_SOURCE is compatible with type INTEGER or INTEGER4.
4. T_DEST is of type INTEGER4 and T_SOURCE is compatible with type INTEGER or WORD.

Also, if T_DEST and T_SOURCE are compatible structured types, then SOURCE is assignment compatible with DESTINATION if one of the following is true:

1. For SETs, every member of SOURCE is in the base type of T_DEST.
2. For LSTRINGs, UPPER (DESTINATION) >= SOURCE.LEN.

Other than in the assignment statement itself, assignment compatibility is required in the following cases of implicit assignment:

1. passing value parameters
2. READ and READLN procedures
3. control variable and limits in a FOR statement
4. super array type array bounds, and array indices

Assignment compatibility is usually known at compile time, and an assignment generates simple instructions. However, some subrange, set, and LSTRING assignments depend on the value of the expression to be assigned and thus cannot be checked until runtime. If the range checking switch is on, assignment compatibility is checked at runtime; otherwise, no checking is done.

Chapter 5

SIMPLE TYPES

The basic distinction between simple and structured data types is that simple types cannot be divided into other types, while structured types (discussed in Chapter 6, "Arrays, Records, and Sets," and Chapter 7, "Files") are composed of other types. The simple data types fall into three categories:

1. ordinal types
2. REAL
3. INTEGER4

Ordinal Types

Ordinal types are all finite and countable. They include the following simple types:

```
INTEGER
WORD
CHAR
BOOLEAN
enumerated types
subrange types
```

INTEGER4, though finite and countable, is not an ordinal type.

INTEGER

INTEGER values are a subset of the whole numbers and range from −MAXINT through 0 to MAXINT. MAXINT is the predeclared constant 32767 (i.e., 2^15 - 1) for current MS-Pascal target machines. (The value −32768 is not a valid INTEGER; the compiler uses it to check for uninitialized INTEGER and INTEGER subrange variables.)

INTEGER is not a subrange of INTEGER4 (discussed in Chapter 5, "INTEGER4"). If it were, signed expressions would have to be calculated using the INTEGER4 type and the result converted to INTEGER.

Expressions are always calculated using a base type, not a subrange type. INTEGER type constants may be changed internally to WORD type if necessary, but INTEGER variables are not. INTEGER values change to REAL or INTEGER4 in an expression, if necessary, but not to REAL. The ORD function converts a value of any ordinal type to an INTEGER type.

The predeclared type INTEGER2 is identical to INTEGER.

WORD

The WORD and INTEGER types are similar, differing chiefly in their range of values. Both are ordinal types. You can think of WORD values as either a group of 16 bits or as a subset of the whole numbers from 0 to MAXWORD (65535, i.e., 2^16 - 1). The WORD type is an MS-Pascal feature that is useful in several ways:

1. to express values in the range from 32768 to 65535
2. to operate on machine addresses
3. to perform primitive machine operations, such as word ANDing and word shifting, without using the INTEGER type and running into the −32768 value

Unlike INTEGERs, all WORDs are nonnegative values. The WRD function changes any ordinal type value to WORD type. Like INTEGER values, WORD values in an expression are converted to the INTEGER4 type, if necessary.

Having both an INTEGER and a WORD type permits mapping of 16-bit quantities in either of two ways:

1. as a signed value ranging from −32767 to +32767
2. as a positive value ranging from 0 to 65535.

However, you must not mix WORD and INTEGER values in an expression (although doing so generates a warning rather than an error message). Neither are WORD and INTEGER values assignment compatible.

CHAR

In MS-Pascal, CHAR values are 8-bit ASCII values. CHAR is an ordinal type. All 256-byte values are included in the type CHAR. In addition, SET OF CHAR is supported. Relational comparisons use the ASCII collating sequence.

Although the line-marker character used in TEXT files is not part of the CHAR type in the ISO standard, some target operating systems for MS-Pascal may require the line-marker character to be included (e.g., carriage return).

The CHR function changes any ordinal type value to CHAR type, as long as ORD of the value is in the range from 0 to 255. See Appendix D, "ASCII Character Codes," for a complete listing of the ASCII character set.

BOOLEAN

BOOLEAN is an ordinal type with only two (predeclared) values: FALSE and TRUE. The BOOLEAN type is a special case of an enumerated type, where ORD (FALSE) is 0 and ORD (TRUE) is 1. This means that FALSE < TRUE.

You may redefine the identifiers BOOLEAN, FALSE, and TRUE, but the compiler implicitly uses the former type in Boolean expressions and in IF, REPEAT, and WHILE statements.

No function exists for changing an ordinal type value to a BOOLEAN type value. However, you can achieve this effect with the ODD function for INTEGER and WORD values, or the expression:

```
ORD (value) <> 0
```

Enumerated Types

An enumerated type defines an ordered set of values. These values are constants and are enumerated by the identifiers that denote them.

Examples of enumerated type declarations:

```
FLAGCOLOR = (RED, WHITE, BLUE)
SUITS = (CLUB, DIAMOND, HEART, SPADE)
DOGS = (MAUDE, EMILY, BRENDAN)
```

Every enumerated type is also an ordinal type. Identifiers for all enumerated type constants must be unique within their declaration level.

At the extend level, the READ and WRITE procedures and the ENCODE and DECODE functions operate on values of an enumerated type by treating the actual constant identifier as a string. This means that enumerated values can be read directly.

The ORD function, at the standard level, can be used to change enumerated values into INTEGER values; the WRD function changes enumerated values into WORD values.

The RETYPE function, at system level, can be used to change INTEGER or WORD values to an enumerated type. For example:

```
IF RETYPE (COLOR, I) = BLUE THEN WRITELN ('TRUE BLUE')
```

The values obtained by applying the ORD function to the constants of an enumerated type always begin with zero. Thus, the values obtained for the type FLAGCOLOR, from the example above, are as follows:

```
ORD (RED)   = 0
ORD (WHITE) = 1
ORD (BLUE)  = 2
```

Enumerated types are particularly useful for representing an abstract collection of names, such as names for operations or commands. Modifying a program by adding a new value to an enumerated type is much safer than using raw numbers, since any arrays indexed with the type or sets based on the type are changed automatically.

For example, interactive input of a command might be accomplished by reading the enumerated type identifier that corresponds to a command. Since enumerated types are ordered, comparisons like RED < GREEN may also be useful. At times, access to the lowest and highest values of the enumerated type is useful with the LOWER and UPPER functions, as in the following example:

```
VAR TINT: COLOR;
FOR TINT := LOWER (TINT) TO UPPER (TINT)
   DO PAINT (TINT)
```

Subrange Types

A subrange type is a subset of an ordinal type. The type from which the subset is taken is called the "host" type. Therefore, all subrange types are also ordinal types.

You can define a subrange type by giving the lower and upper bound of the subrange (in that order). The lower bound must not be greater than the upper bound, but the bounds may be equal. The subrange type is frequently used as the index type of an array bound or as the base type of a set. (See Chapter 6, "Arrays, Records, and Sets," for a discussion of arrays and sets.)

Examples of subranges along with their host ordinal type:

Subrange	Host Ordinal
`INTEGER`	`100..200`
`WORD`	`WRD(1)..9`
`CHAR`	`'A'..'Z'`
enumerated type	`RED..YELLOW`

In addition, you may substitute a subrange clause for a list of values in the following circumstances:

1. set constants
2. set constructors
3. CASE statement constants and record variant labels (at the extend level)

Besides using the subrange type in array and set declarations, you can use it to help to guarantee that the value of a variable is within acceptable bounds. If the range checking switch is on during compilation, these bounds are checked at runtime.

For instance, if the logic of a program implies that a variable always has a value from 100 to 999, then declaring it with a subrange causes the compiler to check that the variable is never assigned a value outside this range.

In addition, declaring a subrange type may permit the compiler to allocate less room and use simpler operations. For example, declaring BOTTLES to be the INTEGER subrange 1..100 means that the type can be allocated in eight bits instead of sixteen.

Three subrange types are predeclared:

1. BYTE = WRD(0)..255; {8-bit WORD subrange}
2. SINT = -127..127; {8-bit INTEGER subrange}
3. INTEGER1 = SINT

The BYTE type is particularly useful in machine-oriented applications. For example, the ADRMEM and ADSMEM types (see Chapter 8, "Address Types," for details) normally treat memory as an array of bytes. However, since the BYTE type is really a subrange of the WORD type, expressions with BYTE values are calculated using 16-bit instead of 8-bit arithmetic, if necessary.

In some cases (for example, an assignment of a BYTE expression to a BYTE variable when the math checking switch is off), the compiler can optimize 16-bit arithmetic to 8-bit arithmetic. In general, using BYTE instead of WORD saves memory at the expense of BYTE-to-WORD conversions in expression calculations.

At the extend level, subrange bounds can be constant expressions. Because the compiler assumes that the left parenthesis always starts an enumerated type declaration, the first expression in a subrange declaration must not start with a left parenthesis. For example:

```
TYPE {First two are permitted.}
     FEE = (A, B, C);
     FIE = M + 2 * N .. (P - 2) * N;
     {FOO is invalid as declared.}
     FOO = (M + 2) * N .. P - 2 * N;
```

REAL

REAL values are nonordinal values of a given range and precision; the range of allowable values depends on the target system. Refer to the *Microsoft Pascal Compiler User's Guide* for more specific information about your system.

Most MS-Pascal implementations use either the Microsoft or IEEE single precision real number format. These formats have a 24-bit mantissa and an 8-bit exponent, giving about seven digits of precision and a maximum value of 1.701411E38. Microsoft format REAL constants are limited to the range 1.0E-38 to 1.0E+38.

The current version of MS-Pascal includes expanded numeric data types for processing higher precision real (and integer) numbers. For reals, this includes support for single and double precision real numbers according to the IEEE floating-point standard.

Standard Pascal provides a type REAL. MS-Pascal provides three real types: REAL, REAL4, and REAL8. However, the type REAL is always identical to either REAL4 or REAL8. The choice is made with a metacommand, $REAL:n, where n is either 4 or 8. |$REAL:8| has the same effect as TYPE REAL = REAL8. The default type for REAL is normally REAL4, but may be changed (see Appendix A, "Version Specifics," in the *Microsoft Pascal Compiler User's Guide* for details).

Any or all of these real number forms may be used in a single program. However, programs that use REAL4 and REAL8 will not be portable.

The REAL4 type is in 32-bit IEEE format, and the REAL8 type is in 64-bit IEEE format. The IEEE standard format is as follows:

REAL4	Sign bit, 8-bit binary exponent with bias of 127, 23-bit mantissa
REAL8	Sign bit, 11-bit binary exponent with bias of 1023, 52-bit mantissa

In both cases the mantissa has a "hidden" most significant bit (always one) and represents a number greater than or equal to 1.0 but less than 2.0. An exponent of zero means a value of zero, and the maximum exponent means a value called NaN (not a number). Bytes are in "reverse" order; the lowest addressed byte is the least significant mantissa byte.

The REAL4 numeric range is barely seven significant digits (24 bits), with an exponent range of E-38 to E+38. The REAL8 numeric range includes over fifteen significant digits (53 bits), with an exponent range of E-306 to E+306 (a very large number!)

The exponent character can be "D" or "d" as well as "E" or "e", so a number like 12.34d56 is permitted. This minor extension provides compatibility with other Microsoft languages. However, the D or d exponent character does not indicate double precision (as it does in FORTRAN), since this would imply that numbers with an E or e exponent characters are single precision.

REAL literals in MS-Pascal are converted first to REAL8 format and then to REAL4 as necessary (for example, to be passed as a CONST parameter or to initialize a variable in a VALUE section). If you need actual REAL4 constants, you must declare them as REAL4 variables (perhaps adding the READONLY attribute) and assign them a constant in a VALUE section.

Both REAL4 and REAL8 values are passed to intrinsic functions as reference (CONSTS) parameters, rather than as value parameters. The compiler accepts REAL expressions as CONSTS parameters; it will evaluate the expression, assign the result to a stack temporary, and pass the address of the temporary, which is usually more efficient than passing the value itself (especially in the REAL8 case).

Functions that return REAL values use the long return method; that is, the caller passes an additional, hidden, offset address of a stack temporary which will receive the result. This applies to all functions returning REAL4 or REAL8 values, both user-defined and intrinsic. See Chapter 11, "Boolean Expressions," for a description of REAL comparisons that produce an unordered result.

The MS-Pascal runtime library provides additional REAL functions to support Microsoft(r) FORTRAN. These functions are available in MS-Pascal, but are not predeclared (see Chapter 14, "Available Procedures and Functions," for further information on the functions available and how to use them).

INTEGER4

Like INTEGER and WORD values, INTEGER4 values are a subset of the whole numbers. INTEGER4 values range from -MAXLONG to MAXLONG. MAXLONG is a predeclared constant with the value of 2,147,483,647 (i.e., 2^31 - 1). The value -2,147,487,648 (i.e., -2^31) is not a valid INTEGER4.

Unlike INTEGER and WORD, the INTEGER4 type is not considered an ordinal type. There are no INTEGER4 subranges and INTEGER4 cannot be an array index or the base type of a set. Also, INTEGER4 values cannot be used to control FOR and CASE statements.

INTEGER4 is currently an extended numeric type, like REAL. Values of type INTEGER or WORD in an expression change automatically to INTEGER4 if the expression requires an intermediate value that is out of the range of either INTEGER or WORD. Values of type INTEGER4 do not change to REAL in an expression; you must explicitly use the FLOATLONG function to make the conversion.

Chapter 6

ARRAYS, RECORDS, AND SETS

A structured type is composed of other types. The components of structured types are either simple types or other structured types. A structured type is characterized by the types of its components and by its structuring method. In MS-Pascal, a structured type can occupy up to 65534 bytes of memory.

The structured types in MS-Pascal are the following:

```
ARRAY <range> OF <type>
SUPER ARRAY <range> OF <type>
   STRING (n)
   LSTRING (n)
RECORD
SET OF <base-type>
FILE OF <type>
```

Because components of structures can be structured types themselves, you may have, for example, an array of arrays, a file of records containing sets, or a record containing a file and another record. This is an example of the data typing flexibility that provides Pascal with much of its linguistic power as a computing language.

The remainder of this chapter discusses arrays, records, and sets. See Chapter 7, "Files," for a discussion of files.

Arrays

An array type is a structure that consists of a fixed number of components. All of the components are of the same type (called the "component type").

The elements of the array are designated by indices, which are values of the "index type" of the array. The index type must be an ordinal type: BOOLEAN, CHAR, INTEGER, WORD, subrange, or enumerated.

Arrays in Pascal are one dimensional, but since the component type can also be an array, n-dimensional arrays are supported as well.

Examples of type declarations for arrays:

```
TYPE
INT_ARRAY : ARRAY [1..10] OF INTEGER;
ARRAY_2D : ARRAY [0..7] OF ARRAY [0..8] OF 0..9;
MORAL_RAY : ARRAY [PEOPLE] OF (GOOD, EVIL)
```

In the last declaration, PEOPLE is a subrange type, while GOOD and EVIL are enumerated constants.

A short-hand notation available for n-dimensional arrays makes the following statement the same as the second example in the preceding paragraph:

```
ARRAY_2D : ARRAY [0..7, 0..8] OF 0..9;
```

After declaring these arrays, you could assign values to components of the arrays with statements such as these:

```
INT_ARRAY [10] := 1234;
ARRAY_2D [0,99] := 999;
MORAL_RAY [Machiavelli] := EVIL;
```

All of an n-dimensional PACKED array is packed; therefore these statements are equivalent:

```
PACKED ARRAY [1..2, 3..4] OF REAL
PACKED ARRAY [1..2] OF PACKED ARRAY [3..4] OF REAL
```

See Chapter 8, "Reference and Other Types," for a discussion of packed types.

Super Arrays

A super array is an example of an MS-Pascal "super type." A super type is like a set of types or like a function that returns a type. Super types in general, and super arrays in particular, are features of MS-Pascal.

The super array type has several important uses. You may use them for any of the following purposes:

1. To process strings.

   Both STRING and LSTRING are predeclared super array types. The LSTRING type handles variable length strings. STRING handles fixed-length strings and strings more than 255 characters long.

2. To dynamically allocate arrays of varying sizes.

   Otherwise such arrays would need a maximum possible size allocation.

3. As the formal parameter type in a procedure or function.

   Such a declaration makes the procedure or function usable for a set or class of types, rather than for just a single fixed-length type.

A super type identifier specifies the set of types represented by the super type. A later type declaration may declare a normal type identifier as a type "derived" from that class of types. This derived type is like any other type.

A super array type declaration is an array type declaration prefixed with the keyword SUPER. Every array upper bound is replaced with an asterisk, as shown:

```
TYPE VECTOR = SUPER ARRAY [1..*] OF REAL;
```

Following the preceding type declaration, you could declare the following variables:

```
VAR   ROW: VECTOR (10);
      COL: VECTOR (30);
      ROWP: ^ VECTOR;
```

In this example, VECTOR is a super array type identifier. VECTOR (10) and VECTOR (30) are type designators denoting "derived types." ROW and COL are variables of types derived from VECTOR. ROWP is a pointer to the super array type VECTOR.

Although the general concept of super types allows other "types of types," such as super subranges and super sets (in addition to super arrays), super types currently allow only an array type with parametric upper bounds. A super type is a class of types and not a specific type. Thus, in the VAR section of a program, procedure, or function, you cannot declare the variables to be of a super type; you must declare them as variables of a type derived from the super type.

However, a formal reference parameter in a procedure or function can be given a super type: this allows the routine to operate on any of the possible derived types. (This kind of parameter is called a "conformant array" in other Pascals.)

A pointer referent type can also be given a super type. This allows a pointer to refer to any of the possible derived types. A pointer referent to a super type allows "dynamic arrays." These arrays are allocated on the heap by passing their upper bound to the procedure NEW. See Chapter 8, "Reference and Other Types," for a discussion of pointer types and dynamic allocation. See Chapter 14, "Predeclared Procedures and Functions," for a description of the procedure NEW.

Example using the NEW procedure for dynamic allocation:

```
VAR STR_PNT: ^SUPER PACKED ARRAY [1..*] OF CHAR;
    VEC_PNT: ^SUPER ARRAY [0..*, 0..*] OF REAL;
      .
      .
    NEW (STR_PNT, 12);
    NEW (VEC_PNT, 9, 99);
```

An actual parameter in a procedure or function can be of a super type rather than a derived type, but only if the parameter is a reference parameter or pointer referent. (These are the only kinds of variables that can be of a super rather than a derived type.)

Example of super arrays:

```
TYPE VECTOR = SUPER ARRAY [1..*] OF REAL;
{''VECTOR'' is the super array type identifier.}

VAR X: VECTOR (12); Y: VECTOR (24); Z: VECTOR (36);
{X, Y, and Z are types derived from VECTOR.}

{Below, SUM accepts variables of all types}
{derived from the super type VECTOR.}
FUNCTION SUM (VAR V: VECTOR): REAL;
{V is the formal reference parameter of the}
{super type VECTOR.}

VAR S: REAL; I: INTEGER;
BEGIN
  S := 0;
  FOR I := 1 TO UPPER (V) DO S := S + V [I];
  SUM := S;
END;

BEGIN
  .
  .
  TOTAL := SUM (X) + SUM (Y) + SUM (Z);
  .
  .
END
```

The normal type rules for components of a super array type and for type designators that use a super array type allow components to be assigned, compared, and passed as parameters.

The UPPER function returns the actual upper bound of a super array parameter or referent. The maximum upper bound of a type derived from a super array type is limited to the maximum value of the index type implied by the lower bound (e.g., MAXINT, MAXWORD). Two super array types are predeclared, STRING and LSTRING. The compiler directly supports STRING and LSTRING types in the following ways:

1. LSTRING and STRING assignment
2. LSTRING and STRING comparison
3. LSTRING and STRING READs
4. access to the length of a STRING with the UPPER function
5. access to maximum length of an LSTRING with the UPPER function
6. access to LSTRING length with STR.LEN and STR[0]

These subjects are discussed later in Chapter 6, "Using STRINGs AND LSTRINGs."

STRINGs

STRINGs are predeclared super arrays of characters:

```
TYPE STRING = SUPER PACKED ARRAY [1..*] OF CHAR;
```

A string literal such as 'abcdefg' automatically has the type STRING (n). The size of the array 'abcdefg' is 7; thus, the constant is of the STRING derived type, STRING (7).

Standard Pascal calls any packed array of characters with a lower bound of one a "string" and permits a few special operations on this type (such as comparison and writing) that you cannot do with other arrays.

In MS-Pascal, the super array notation STRING (n) is identical to PACKED ARRAY [1..n] OF CHAR (n may range from 1 to MAXINT). There is no default for n, as in some other Pascals, since STRING means the super array type itself and not a string with a default length.

The identifier STRING is for a super array, so you can only use it as a formal reference parameter type or pointer referent type. The other super array restrictions apply: you may not compare such a parameter or dereferenced pointer or assign it as a whole.

Any variable (or constant) with the super array type STRING, or one of the types CHAR or STRING (n) or PACKED ARRAY [1..n] OF CHAR, can be passed to a formal reference parameter of super array type STRING. Furthermore, a variable of type LSTRING or LSTRING (n) can also be passed to a formal reference parameter of type STRING. For a discussion of STRING as a formal reference parameter, see Chapter 6, "Using STRINGs and LSTRINGs."

Standard Pascal supports the assigning, comparing, and writing of STRINGs. The extend level permits reading STRINGs, including the super array type STRING and a derived type STRING (n). Reading a STRING causes input of characters until the end of a line or the end of the STRING is reached. If the end of the line is reached first, the rest of the STRING is filled with blanks. Writing a string writes all of its characters.

The normal Pascal type compatibility rules are relaxed for STRINGs. Any two variables or constants with the type PACKED ARRAY [1..n] OF CHAR or the type STRING (n) can be compared or assigned if the lengths are equal. However, since the length of a STRING super array type may vary, comparisons and assignments are not allowed.

Example of an illegal STRING assignment:

```
PROCEDURE CANNOT_DO (VAR S : STRING);
VAR STR : STRING (10);
BEGIN
  STR := S
 {This assignment is illegal because}
 {the length of S may vary.}
END;
```

The PACKED prefix in the declaration PACKED ARRAY [1..n] OF CHAR, as defined in the ISO standard, normally implies that a component cannot be passed as a reference parameter. In MS-Pascal, this restriction does not apply.

To keep conformance to the ISO standard, this passing of the CHAR component of a STRING as a reference parameter is defined as an "error not caught." Also, the index type of a string is officially INTEGER, but WORD type values can also be used to index a STRING. Many string-processing applications are expected to take advantage of the LSTRING type, described in Chapter 6, "LSTRINGs."

A number of intrinsic procedures and functions for strings are discussed in Chapter 14, "Available Procedures and Functions." Many of the procedures and functions described work on STRINGs; some apply only to LSTRINGs.

LSTRINGs

The LSTRING feature in MS-Pascal allows variable-length strings. LSTRING (n) is predeclared as:

```
TYPE LSTRING = SUPER PACKED ARRAY [0..*] OF CHAR
```

However, a variable with the explicit type PACKED ARRAY [0..n] OF CHAR is not "identical" to the type LSTRING (n) even though they are structurally the same. There is no default for n; the range of n is from zero to 255. Characters in an LSTRING can be accessed with the usual array notation.

Internally, LSTRINGs contain a length (L), followed by a string of characters. The length is contained in element zero of the LSTRING and can vary from zero to the upper bound. The length of an LSTRING variable T can be accessed as T[0] with type CHAR, or as T.LEN with type BYTE. String constants of type CHAR or STRING (n) are changed automatically to type LSTRING.

The predeclared constant NULL is the empty string, LSTRING (0). NULL is the only constant with type LSTRING; there is no way to define other LSTRING constants. As with STRINGs, a CHAR component of an LSTRING can be passed as a reference parameter, and WORD and INTEGER values can be used to index an LSTRING.

Several operations work differently on LSTRINGs than on STRINGs. Any LSTRING can be assigned to any other LSTRING, so long as the current length of the right side is not greater than the maximum length of the left side. Similarly, an LSTRING can be passed as a value parameter to a procedure or function, so long as the current length of the actual parameter is not greater than the maximum length specified by the formal parameter. If the range checking switch is on, the compiler checks the assignment of LSTRINGs and the passing of LSTRING (n) parameters. The actual number of bytes assigned or passed is the minimum of the upper bounds of the LSTRINGs.

Neither side in an LSTRING assignment can be a parameter of the super array type LSTRING; both must be types derived from it.

Examples of LSTRING assignments:

```
 {Declaring the variables}
 VAR A : LSTRING (19);
     B : LSTRING (14);
     C : LSTRING (6);
     .
     .
 {Assigning the variables}
 A := '19 character string';
 B := '14 characters';
 C := 'shorty';
 A := B;
{This is legal, since the length of B}
{is less than the maximum length of A.}
C := A;
{This is illegal, since length of A}
{is greater than the maximum length of C.}
```

You may compare any two LSTRINGs, including super array type LSTRINGs (the only super array type comparison allowed). Reading an LSTRING variable causes input of characters, until the end of the current line or the end of the LSTRING, and sets the length to the number of characters read. Writing from an LSTRING writes the current length string.

Using STRINGs and LSTRINGs

This section describes the STRING and LSTRING operations directly supported by the compiler. An annotated program at the end of this section illustrates the use of STRINGs and LSTRINGs in context.

See also Chapter 14, "Predeclared Procedures and Functions," for descriptions of the following string procedures and functions:

```
CONCAT      INSERT
COPYLST     POSITN
COPYSTR     SCANEQ
DELETE      SCANNE
```

At the system level of MS-Pascal, the procedures FILLC, FILLSC, MOVEL, MOVESL, MOVER, and MOVESR also operate on strings.

MS-Pascal supports STRINGs and LSTRINGs directly in the following ways:

1. Assignment

   You may assign any LSTRING value to any LSTRING variable, as long as the maximum length of the target variable is greater than or equal to the current length of the source value and neither is the super array type LSTRING. If the maximum length of the target is less than the current length of the source, only the target length is assigned, and a runtime error occurs if the range checking switch is on. You may assign a STRING value to a STRING variable, as long as the length of both sides is the same and neither side is the super array type STRING. Passing either STRING or LSTRING as a value parameter is much like making an assignment.

2. Comparison

   The LSTRING operators $<$, $<=$, $>$, $>=$, $<>$, and $=$ use the length byte for string comparisons; the operands may be of different lengths. Two strings must be the same length to be considered equal. If two strings of different lengths are equal up to the length of the shorter one, the shorter is considered less than the longer one. The operands can be of the super array type LSTRING. For STRINGs, the same relational operators are available, but the lengths must be the same and operands of the super array type STRINGs are not allowed.

3. READs and WRITEs

   READ LSTRING reads until the LSTRING is filled or until the end-of-line is found. The current length is set to the number of characters read. WRITE LSTRING uses the current length. See also READSET (described in Chapter 15, "File-Oriented Procedures and Functions"), which reads into an LSTRING as long as input characters are in a given SET OF CHAR. READ STRING pads with spaces if the line is shorter than the STRING. WRITE STRING writes all the characters in the string. Both READ and WRITE permit the super array types STRING and LSTRING, as well as their derived types.

4. Length access

   You can access the current length of an LSTRING variable T with T.LEN, which is of type BYTE, or with T[0], which is of type CHAR. This notation can assign a new length, as well as determine the current length. The UPPER function will find the maximum length of an LSTRING or the length of a STRING. This is especially useful for finding the upper bound of a super array reference parameter or pointer referent.

You cannot assign or compare mixed STRINGs and LSTRINGs, unless the STRING is constant. You can assign STRINGs to LSTRINGs, or vice versa, with one of the move routines or with the COPYSTR and COPYLST procedures. Since constants of type STRING or CHAR change automatically to type LSTRING if necessary, LSTRING constants are considered normal STRING constants. NULL (the zero length LSTRING) is the only explicit LSTRING constant.

In the sample program at the end of this section, all STRING parameters (CONST or VAR) may take either a STRING or an LSTRING; all LSTRING parameters are VAR LSTRING and must take an LSTRING variable.

A "special transformation" lets you pass an actual LSTRING parameter to a formal reference parameter of type STRING. The length of the formal STRING is the actual length of the LSTRING. Therefore, if LSTR (in the following example) is of type LSTRING (n) or LSTRING, it can be passed to a procedure or function with a formal reference parameter of type STRING:

```
VAR LSTR : LSTRING (10);
   .
   .
PROCEDURE TIE_STRING (VAR STR : STRING);
   .
   .
TIE_STRING (LSTR);
```

In this case, UPPER (STR) is equivalent to LSTR.LEN.

Procedures and functions with reference parameters of super type STRING can operate equally well on STRINGs and LSTRINGs. The only reason to declare a parameter of type LSTRING is when the length must be changed. Normally, an LSTRING is either a VAR or a VARS parameter in a procedure or function, since a CONST or CONSTS parameter of type LSTRING cannot be changed.

Example of a program that uses STRINGs and LSTRINGs:

```
PROGRAM STRING_SAMPLE;
PROCEDURE STRING_PROC (CONST S: STRING); BEGIN END;
PROCEDURE LSTRING_PROC (CONST S: LSTRING); BEGIN END;

VAR
  CHR1VAR: CHAR;
  STR5VAR: STRING (5);
  LST5VAR: LSTRING (5);
  LST9VAR: LSTRING (9);
  STR4VAR: PACKED ARRAY [1..4] OF CHAR;
  STR6VAR: PACKED ARRAY [1..6] OF CHAR;

BEGIN
{Look at all the kinds of strings a}
{CONST STRING parameter takes.}
STRING_PROC ('A');
{Character constant is OK.}
STRING_PROC (CHR1VAR);
{Character variable is OK.}
STRING_PROC ('STRING');
{STRING constant is OK.}
STRING_PROC (STR5VAR);
{STRING variable is OK.}
STRING_PROC (LST5VAR);
{LSTRING variable is OK.}
```

```
{However, a CONST LSTRING parameter cannot take}
{non-LSTRING variables.}
LSTRING_PROC ('A');
{Character constant is OK.}
LSTRING_PROC (CHR1VAR);
{Character variable is not OK!}
LSTRING_PROC ('STRING');
{STRING constant is OK.}
LSTRING_PROC (STR5VAR);
{STRING variable is not OK!}
LSTRING_PROC (LST5VAR);
{LSTRING variable is OK.}
{Assignments to a STRING variable are limited}
{to the same type.}
STR5VAR := 'A';
{Character constant is not OK!}
STR5VAR := CHR1VAR;
{Character variable is not OK!}
STR5VAR := 'TINY';
{STRING constant too small.}
STR5VAR := 'RIGHT';
{Both sides have five characters; OK.}
STR5VAR := 'longer';
{Not OK; STRING constant is too large.}
STR5VAR := LST5VAR;
{Not OK; you cannot assign LSTRINGs to STRINGs.}
COPYSTR (LST5VAR, STR5VAR);
{COPYSTR is an intrinsic procedure.}
STR5VAR := STR4VAR;
{Not OK; STRING variable is too small.}
COPYSTR (STR4VAR, STR5VAR);
{COPYSTR is OK; padding of space in STR5VAR[5].}
STR5VAR := STR5VAR;
{OK; both sides have five characters.}
STR5VAR := STR6VAR;
{Not OK; STRING variable is too large.}
```

```
{Assignments to an LSTRING variable, however,}
{are more flexible.}
LST5VAR := 'A';
{Character constant is OK.}
LST5VAR := CHR1VAR;
{Character variable is not OK!}
LST5VAR := 'TINY';
{Smaller STRING constant is OK.}
LST5VAR := 'RIGHT';
{Same length STRING constant is OK.}
LST5VAR := 'LONGER';
{This gives an error at runtime only; OK for now.}
LST5VAR := LST9VAR;
{This may give an error at runtime; OK for now.}
LST9VAR := LST5VAR;
{This isn't even checked at runtime; always OK.}
LST5VAR := STR5VAR;
{Not OK; you cannot assign a STRING variable to an}
{LSTRING variable.}
COPYLST (STR5VAR, LST5VAR);
{This is the way to copy a STRING variable}
{to an LSTRING.}

END.
```

Records

A record structure acts as a template for conceptually related data of different types. The record type itself is a structure consisting of a fixed number of components, usually of different types.

Each component of a record type is called a field. The definition of a record type specifies the type and an identifier for each field within the record. Because the scope of these "field identifiers" is the record definition itself, they must be unique within the declaration. The field values associated with field identifiers are accessible with record notation or with the WITH statement.

For example, you could declare the following record type:

```
TYPE LP = RECORD
          TITLE : LSTRING (100);
          ARTIST : LSTRING (100);
          PLASTIC : ARRAY
             [1..SONG_NUMBER] OF SONG_TITLE
          END
```

You could then declare a variable of the type LP, as follows:

```
VAR BEATLES_1 : LP;
```

Finally, you could access a component of the record with either field notation or the WITH statement (note the period separating field identifiers):

```
BEATLES_1.TITLE := 'Meet The Beatles';
WITH BEATLES_1 DO
   PLASTIC[1] := 'I Wanna Hold Your Hand'
```

Variant Records

A record may have several "variants," in which case a certain field called the "tag field" indicates which variant to use. The tag field may or may not have an identifier and storage in the record. Some operations, such as the NEW and DISPOSE procedures and the SIZEOF function, can specify a tag value even if the tag is not stored as part of the record.

Examples of variant records:

```
TYPE OBJECT = RECORD
       X, Y: REAL;
       CASE S: SHAPE OF
          SQUARE: (SIZE, ANGLE: REAL);
          CIRCLE: (DIAMETER: REAL)
     END;
     FOO = RECORD    .
       CASE BOOLEAN OF
         TRUE: (I, J: INTEGER);
         FALSE: (CASE COLOR OF
                BLUE: (X: REAL);
                RED:  (Y: INTEGER4));
     END;
```

Only one variant part per record is allowed; it must be the last field of the record. However, this variant part can also have a variant (and so on, to any level). All field identifiers in a given record type must be unique, even in different variants. For example, after declaring the record types above, you could create and then assign to the variables shown in the following example:

```
VAR O, P : OBJECT;
    F, G : FOO;
BEGIN
  O.DIAMETER := 12.34; {CASE of CIRCLE}
  P.SIZE := 1.2;       {CASE of SQUARE}
  F.I := 1; F.J := 2;  {CASE of TRUE}
  G.X := 123.45;       {CASE of FALSE and BLUE}
  G.Y := 678999        {CASE of FALSE and RED;}
                       {this overwrites G.X.}
END;
```

The latest ISO standard requires every possible tag field value to select some variant. Therefore, it is illegal to include CASE INTEGER OF and omit a variant for every possible INTEGER value. However, such an omission is an error not caught in MS-Pascal.

MS-Pascal supports the use of full CASE constant options in the variant clause; that is, a list of constants can define a case. At the extend level, subranges and the OTHERWISE statement can also define a case. If used, OTHERWISE applies to the last variant in the list and is not followed by a colon. You can also declare an empty variant, such as POINT:() or OTHERWISE (). You can even declare an entirely empty record type, although the compiler issues a warning whenever the record is used.

The ISO standard defines a number of errors that relate to variant records; these errors may not be caught in MS-Pascal, even if the tag-checking switch is on. (The tag-checking switch generates code each time a variant field is used, to check that the tag value is correct.) In the record type declaration of OBJECT (in the previous example), any use of SIZE generates a check that S = SQUARE. However, in the case of FOO, uses of "I" cannot be checked because MS-Pascal does not allocate the BOOLEAN tag field.

The ISO standard further declares that when a "change of variant" occurs (such as when a new tag value is assigned), all the variant fields become undefined. However, MS-Pascal does not set the fields to an uninitialized value when a new tag is assigned. Therefore, using a variant field with an undefined value is an error not caught in MS-Pascal.

Nor does MS-Pascal enforce various restrictions on a record variable allocated on the heap with the long form of the NEW procedure (see Chapter 14, "Available Procedures and Functions," for details). However, MS-Pascal does check an assignment to such a "short record" to see that only the short record itself is modified in the heap.

A record allocated with the long form of NEW may be released using the short form of DISPOSE with no ill effects (this is an ISO error not caught in MS-Pascal). It is also an error not caught in MS-Pascal to DISPOSE of a record passed as a reference parameter or used by an active WITH statement.

Variant records interact with MS-Pascal features in two ways:

1. Declaring a variant that contains a file is not safe; any change to the file's data using a field in another variant may lead to I/O errors, even if the file is closed. In the following example, any use of R will lead to errors in F:

```
RECORD CASE BOOLEAN OF
    TRUE : (F: FILE OF REAL);
    FALSE : (R: ARRAY [1..100] OF REAL);
    END;
```

2. Giving initial data to several overlapping variants in a variable in a VALUE section could have unpredictable results. In the following example, the initial value of LAP is uncertain:

```
VAR LAP : RECORD CASE BOOLEAN OF
    TRUE : (I: INTEGER4);
    FALSE : (R: REAL);
    END;
VALUE LAP.I := 10; LAP.R := 1.5;
```

MS-Pascal generates a warning message if you attempt either of these operations.

Explicit Field Offsets

MS-Pascal lets you assign explicit byte offsets to the fields in a record. This system level feature can be useful for interfacing to software in other languages, since control block formats may not conform to the usual MS-Pascal field allocation method. However, because it also permits unsafe operations, such as overlapping fields and word values at odd byte boundaries, it is not recommended unless the interface is necessary.

Example showing assignment of explicit byte offsets:

```
TYPE CPM = RECORD
              NDRIVE [00]: BYTE;
              FILENM [01]: STRING (8);
              FILEXT [09]: STRING (3);
              EXTENT [12]: BYTE;
              CPMRES [13]: STRING (20);
              RECNUM [33]: WORD;
              RECOVF [35]: BYTE;
            END;
     OVERLAP = RECORD
              BYTEAR [00]: ARRAY [0..7] OF BYTE;
              WORDAR [00]: ARRAY [0..3] OF WORD;
              BITSAR [00]: SET OF 0..63;
            END;
```

As may be seen in the example, the offset is enclosed in brackets; this is similar to attribute notation. The number is the byte offset to the start of the field. Some target machines may not permit accessing a 16-bit value at an odd address, but the compiler doesn't catch this as an error.

If you give any field an offset, give offsets to all fields. For any offset that you omit, the compiler picks an arbitrary value. Although the compiler will process a declaration that includes both offsets and variant fields, you should use only one or the other in a given program.

Although you can completely control field overlap with explicit offsets, variants provide the long forms of the procedures NEW, DISPOSE, and SIZEOF. If you want to allocate different length records, use the RETYPE and GETHQQ procedures, instead of variants and the long form of NEW. For example:

```
CPMPV := RETYPE (CPMP, GETHQQ (36));
```

The compiler does support structured constants for record types with explicit offsets. Internally, odd length fields greater than one are rounded to the next even length. For example:

```
ODDR = RECORD
          F1[00] : STRING (3);
          F2[03] : CHAR
        END;
```

In this example, field F1 is four bytes long, so an assignment to F1 overwrites F2. In such a record, all odd length fields must be assigned first.

Sets

A set type defines the range of values that a set may assume. This range of assumable values is the "power set" of the base type you specify in the type definition. The power set is the set of all possible sets that could be composed from an ordinal base type. The null set, [], is a member of every set.

Suppose you declare the following set types:

```
TYPE HUES = SET OF COLOR;
     CAPS = SET OF 'A'..'Z';
     MATTER = SET OF (ANIMAL, VEGETABLE, MINERAL);
```

Then you declare variables like the following:

```
VAR FLAG : HUES;
    VOWELS : CAPS;
    LIVE : MATTER;
```

Finally, you could assign these set variables:

```
FLAG := [RED, WHITE, BLUE];
VOWELS := ['A', 'E', 'I', 'O', 'U'];
LIVE := [ANIMAL, VEGETABLE];
```

The set elements must be enclosed in brackets. This practice differs from the use of parentheses to enclose the base enumerated type in a set type declaration.

Set operations are implemented directly by generated in-line code or by routines in the set unit. See Chapter 11, "Expressions," for a complete discussion of operations on sets.

The ORD value of the base type can range from 0 to 255. Thus, SET OF CHAR is legal, but SET OF 1942..1984 is not.

Sets whose maximum ORD value is 15 (i.e., sets that fit into a WORD) are usually more efficient than larger ones. Also, if the range checking switch is on, passing a set as a value parameter invokes a runtime compatibility check, unless the formal and actual sets have the same type.

Sets provide a clear and efficient way of giving several qualities or attributes to an object. In another language, you might assign each quality a power of two:

```
READY = 1
GETSET = 2
ACTIVE = 4
DONE = 8
```

You might then assign the qualities with a statement like this:

```
X := READY + ACTIVE
```

and then test them using OR and AND as bitwise operators with a statement like:

```
IF ((X AND ACTIVE) <> 0) THEN WRITELN ('GO FISH')
```

The equivalent declaration in MS-Pascal might be:

```
QUALITIES = SET OF (READY, GETSET, ACTIVE, DONE);
```

You could then assign the qualities with X := [GETSET, ACTIVE] and test them with the following operations:

IN	tests a bit
+	sets a bit
-	clears a bit

For example, an appropriate construction might be:

```
IF ACTIVE IN X THEN WRITELN ('GO FISH')
```

You can also use SET OF 0..15 to test and set the bits in a WORD. Using WORDs both as a set of bits and as the WORD type requires giving two types to the word, with a variant record, the RETYPE function, or an address type.

The bits in a set are assigned starting with the most significant bit in the lowest addressed byte. Thus, on a byte-swapped machine, the set [0, 7, 8, 15] has the WORD value #80 + #01 + #8000 + #0100. See the *Microsoft Pascal Compiler User's Guide* for further details.

Chapter 7

FILES

A file is a structure that consists of a sequence of components, all of the same type. It is through files that Microsoft Pascal interfaces with a given operating system. Therefore, you must understand the FILE type in order to perform input to and output from a program.

Declaring Files

As with any other type, you must declare a file variable in order to use it. However, the number of components in a file is not fixed by declaring a FILE type.

Examples of FILE declarations:

```
TYPE F1 = FILE OF COLOR;
     F2 = FILE OF CHAR;
     F3 = TEXT;
```

Conceptually, a file is simply another data type, like an array, but with no bounds and with only one component accessible at a time. However, a file usually corresponds to one of the following:

1. disk files
2. terminals
3. printers
4. other input and output devices

This implies the following restriction in Pascal: a FILE OF FILE is illegal, directly or indirectly. Other structures, such as a FILE OF ARRAYs or an ARRAY OF FILEs, are permitted.

Most Pascal implementations connect file variables to the data files of the operating system. MS-Pascal always uses the target operating system to access files but does not impose additional formatting or structure on operating system files.

MS-Pascal supports normal statically allocated files, files as local variables (allocated on the stack), and files as pointer referents (allocated on the heap). Except for files in super arrays, the compiler generates code to initialize a file when it is allocated and to CLOSE a file when it is deallocated.

This initialization call occurs automatically in most cases. However, a file declared in a module or uninitialized unit's interface will only get its initialization call if you call the module or unit identifier as a procedure. File declarations in such cases get the following compiler warning:

```
Contains file initialize module
```

Only a file in an interface of an uninitialized unit does not generate this warning.

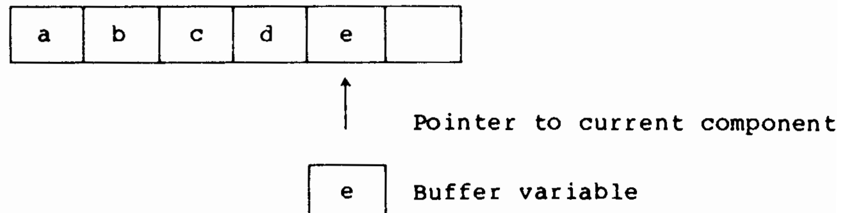
MS-Pascal sets up the standard files, INPUT and OUTPUT (discussed in Chapter 7, "The Predeclared Files, INPUT and OUTPUT"). In standard Pascal, files must be given in the program header, and when you run your program, the runtime system prompts you for filenames. At the extend level, you may use the ASSIGN and READFN procedures to give explicit operating system filenames to files not included in the program header.

Files in record variants or super array types are not recommended; if you use them, the compiler issues a warning. A file variable cannot be assigned, compared, or passed by value: it can only be declared and passed as a reference parameter.

At the extend level, you may indicate a file's access method or other characteristics by specifying the mode of the file. The mode is a value of the predeclared enumerated type FILEMODES. The modes available normally include the three base modes, SEQUENTIAL, TERMINAL, and DIRECT. All files, except INPUT and OUTPUT, are given SEQUENTIAL mode by default. INPUT and OUTPUT are given the default mode TERMINAL.

The Buffer Variable

Every file F has an associated buffer variable F^. A buffer variable and its associated file might look like this:



The procedures GET and PUT use this buffer variable to READ from and WRITE to files. GET copies the current component of the file to the buffer variable. PUT does the opposite; that is, PUT copies the value of the buffer variable to the current component.

The buffer variable can be referenced (i.e., its value fetched or stored) like any other MS-Pascal variable. This allows execution of assignments like the following:

```
F^ := 'z'
C := F^
```

A file buffer variable can be passed as a reference parameter to a procedure or function or used as a record in a WITH statement. However, the file buffer variable may not be updated correctly if the file position changes within the procedure, function, or WITH statement. The compiler issues a warning message to alert you to this possibility.

For example, the following use of a file buffer variable would generate a warning at compile time:

```
VAR A : TEXT;
PROCEDURE CHAR_PROC (VAR X : CHAR);
.
.
CHARPROC (A^);
{Warning issued here}
```

Two special internal mechanisms in MS-Pascal, lazy evaluation and concurrent I/O, allow, respectively, interactive terminal input in a natural way and overlapped I/O along with program execution. Lazy evaluation is applied to all ASCII structured files and is necessary for natural terminal input. Concurrent I/O is applied to all BINARY structured files and is necessary for some operating systems that support overlapping input and output.

Both mechanisms generate a runtime call that is executed before any use of the buffer variable. See "Lazy Evaluation," and "Concurrent I/O," in Chapter 15 for complete details.

File Structures

MS-Pascal files have two basic structures: BINARY and ASCII. These two structures correspond to raw data files and human-readable textfiles, respectively.

BINARY Structure Files

The Pascal data type FILE OF <type> corresponds to MS-Pascal BINARY structure files. These, in turn, correspond to unformatted operating system files.

Under operating systems that divide files into records, every record is one component of the file type (not to be confused with the record type). Primitive procedures such as GET and PUT operate on a record basis. Under operating systems that do not have their own record structure, the primitive procedures GET and PUT transfer a fixed number of bytes per call, equal to the length of one component. See Chapter 7, "File Access Modes," for further discussion of BINARY files.

ASCII Structure Files

The Pascal data type TEXT corresponds to MS-Pascal ASCII structure files. These, in turn, correspond to textual operating system files (called "textfiles" in this manual).

The Pascal TEXT type is like a FILE OF CHAR, except that groups of characters are organized into "lines" and, to a lesser extent, "pages." Primitive file procedures, such as GET and PUT, always operate on a character basis.

However, under operating systems that divide files into records, every record is a line (not a character). Even in operating systems that do not have their own record structure, other languages and utilities have some way of organizing bytes into lines of characters.

MS-Pascal provides a number of special functions and procedures that use this line-division feature. Because MS-Pascal does not impose any additional formatting on operating system files of most modes (including SEQUENTIAL, TERMINAL, and DIRECT), programs in other languages can generate and use these files.

Pascal textfiles (files of type TEXT) are divided into lines with a "line marker," conceptually a character not of the type CHAR. In theory, a textfile can contain any value of type CHAR. However, under some operating systems, writing a particular character (say, CHR (13), carriage return, or CHR (10), line feed) may terminate the current line (record). This character value is the line marker in this case and, when read, always looks like a blank.

Under other operating systems, there may not be a terminating character. Still, as far as you are concerned, every line is followed by a line marker that reads as a blank.

At the extend level, a declaration for a textfile may include an optional line length. Setting the line length, which sets record length, is only needed for DIRECT mode textfiles. You may specify line length for other modes as well, but doing so has no effect.

Specify the line length of a textfile as a constant in parentheses after the word TEXT:

```
TYPE NAMEADDR = TEXT (128);
      DEFAULTX = TEXT;
      SMALLBUF = TEXT (2);
```

File Access Modes

The file modes in MS-Pascal are SEQUENTIAL, TERMINAL, and DIRECT. SEQUENTIAL and TERMINAL mode files are available at the standard level; all three, including DIRECT mode, are available at the extend level. SEQUENTIAL and TERMINAL mode ASCII structure files can have variable length records (lines); DIRECT mode files must have fixed length records or lines.

The declaration of a file in Pascal implies its structure, but not its mode. For example, FILE OF STRING (80) indicates BINARY structure; TEXT indicates ASCII structure. An assignment like F.MODE := DIRECT sets the mode; this only works at the extend level and is currently only needed to set DIRECT mode.

TERMINAL Mode Files

TERMINAL mode files always correspond to an interactive terminal or printer. TERMINAL mode files, like SEQUENTIAL mode files, are opened at the beginning of the file for either reading or writing. Records are accessed one after the other until the end of the file is reached.

Operation of TERMINAL mode input for terminals depends on the file structure (ASCII or BINARY). For ASCII structure (type TEXT), entire lines are read at one time. This permits the usual operating system intraline editing, including backspace, advance cursor, and cancel. Characters are echoed to the terminal screen while the line is being typed.

If the target operating system does not support intraline editing or echo, the MS-Pascal file system interface provides it. However, since an entire line is read at once, you cannot read the characters as you type them, invoke several prompts and responses on the same line, and so on.

For BINARY structure TERMINAL mode (usually type FILE OF CHAR), you can read characters as you type them. No intraline editing or echoing is done. This method permits screen editing, menu selection, and other interactive programming on a keystroke rather than line basis.

TERMINAL mode files use lazy evaluation to properly handle normal interactive reading of the terminal keyboard. See Chapter 15, "Lazy Evaluation," for details.

SEQUENTIAL Mode Files

SEQUENTIAL mode files are generally disk files or other sequential access devices. Like TERMINAL mode files, SEQUENTIAL mode files are opened at the beginning of the file for either reading or writing, and records are accessed one after another until the end of the file. Standard Pascal files are in SEQUENTIAL mode by default (except for INPUT and OUTPUT).

DIRECT Mode Files

DIRECT mode files are generally disk files or other random access devices. DIRECT mode files and the ability to access the mode of a file are available at the extend level of MS-Pascal.

DIRECT mode ASCII structure files, as well as all BINARY structure files, have fixed-length records, where a record is either a line or file component. (Here the term "record" refers not to the normal Pascal record type, but to a disk structuring unit.) DIRECT files are always opened for both reading and writing, and records can be accessed randomly by record number. There is no record number zero; records begin with record number one.

The Predeclared Files INPUT and OUTPUT

Two files, INPUT and OUTPUT, are predeclared in every MS-Pascal program. These files get special treatment as program parameters and are normally required as parameters in the program heading:

```
PROGRAM ACTION (INPUT, OUTPUT);
```

If there are no program parameters and the program does not use the files INPUT and OUTPUT, the heading can look like this:

```
PROGRAM ACTION;
```

However, you should include INPUT and OUTPUT as program parameters if you use them, either explicitly or implicitly, in the program itself:

```
WRITE (OUTPUT, 'Prompt: ') {Explicit use}
WRITE ('Prompt: ')         {Implicit use}
```

These examples would generate a warning if OUTPUT was not declared in the program heading. The only effect of INPUT and OUTPUT as program parameters is to suppress this warning.

Although you may redefine the identifiers INPUT and OUTPUT, the file assumed by textfile input and output procedures and functions (e.g., READ, EOLN) is the predeclared definition. The procedures RESET (INPUT) and REWRITE (OUTPUT) are generated automatically, whether or not INPUT and OUTPUT are present as program parameters (you may also use these procedures explicitly).

INPUT and OUTPUT have ASCII structure and TERMINAL mode. They are initially connected to your terminal and opened automatically. At the extend level of MS-Pascal, you can change these characteristics if you wish.

Extend Level I/O

A file variable in MS-Pascal is really a record, of type FCBFQQ, called a file control block. At the extend level, a few standard fields in this record help you handle file modes and error trapping.

Additional fields and the record type FCBFQQ itself can be used at the system level, described in Chapter 7, "System Level I/O." Along with access to certain FCB fields, extend level I/O also includes the following procedures:

```
ASSIGN      READFN
CLOSE       READSET
DISCARD     SEEK
```

See Chapter 15, "Extend Level I/O," for a description of these procedures.

Use the normal record field syntax to access FCB fields. For a file F, the fields are named F.MODE, F.TRAP, and F.ERRS. You may change or examine these fields at any time.

1. `F.MODE: FILEMODES`

   This field contains the mode of the file: SEQUENTIAL, TERMINAL, or DIRECT. These values are constants of the predeclared enumerated type FILEMODES. The file system uses the MODE field only during RESET and REWRITE. Thus, changing the MODE field of an open file has no effect and is, in fact, discouraged. Except for INPUT and OUTPUT, which have TERMINAL mode, a file's mode is SEQUENTIAL by default.

RESET and REWRITE change the mode from SEQUENTIAL to TERMINAL if they discover that the device being opened is a terminal or printer and if the target operating system allows it. This is useful in programs designed to work either interactively or in batch mode. You must set DIRECT mode before RESET or REWRITE if you plan to use SEEK on a file.

2. `F.TRAP: BOOLEAN`

   If this field is TRUE, error trapping for file F is turned on. Then, if an input/output error occurs, the program does not abort and the error code can be examined. Initially, F.TRAP is set FALSE. If FALSE and an I/O error occurs, the program aborts.

3. `F.ERRS: WRD(0)..15`

   This field contains the error code for file F. An error code of zero means no error; values from 1 to 15 imply an error condition. If you attempt a file operation other than CLOSE or DISCARD and F.ERRS is not zero, the program immediately aborts if F.TRAP is FALSE. However, if F.TRAP is TRUE, the attempted file operation is ignored and the program continues.

   CLOSE and DISCARD do not examine the initial value of F.ERRS, so they are never ignored and do not cause an immediate abort. Nevertheless, if CLOSE or DISCARD themselves generate an error condition, F.TRAP is used to determine whether to trap the error or to abort.

   An operation ignored because of an error condition does not change the file itself, but may change the buffer variable or READ procedure input variables. See Appendix H, "Messages," for a complete listing of error messages and warnings.

Also at the extend level, you may set the line length for a textfile, as shown:

```
TYPE SMALLBUF = TEXT (16);
VAR RANDOMTEXT: TEXT (132);
```

Declaring line length applies only to DIRECT mode ASCII structure files, where the line length is the record length used for reading and writing. Setting the line length has no effect on other ASCII files.

System Level I/O

At the system level of MS-Pascal, you can call procedures and functions that have a formal reference parameter of type FCBFQQ with an actual parameter of the type FILE OF <type> or TEXT, or the identical FCBFQQ type.

The FCBFQQ type is the underlying record type used to implement the file type in MS-Pascal. The interface for the target system FCBFQQ type (and any other types needed) is usually part of the internal file system. Thus, procedures and functions that reference FCBFQQ parameters can be called with any file type, including predeclared procedures and functions like CLOSE and READ.

An FCBFQQ type variable can be passed to procedures like READLN and WRITELN that require a textfile. This permits, for example, calling directly the interface routines on the target operating system, working with mixtures of MS-Pascal and MS-FORTRAN (which share the file system interface but have special FCBFQQ fields), and other special file system activities.

Such activities require a sound knowledge of the file system. See Chapter 8, "An Overview of the File System," in the *Microsoft Pascal Compiler User's Guide* for a discussion of the file system interface and file control block.

Chapter 8

REFERENCE AND OTHER TYPES

The array, record, and set types discussed in Chapter 6 let you describe data structures whose form and size are predetermined and whose components are accessed in a standard way. The file type, described in Chapter 7, "Files," is a structure that varies in size but whose form and means of access are predetermined.

In this chapter, you will find a discussion of reference types, which allow data structures that vary in size and form and whose means of access is particular to the programming problem involved. Also included are notes on PACKED types and procedural and functional types.

Reference Types

A reference to a variable or constant is an indirect way to access it. The pointer type is an abstract type for creating, using, and destroying variables allocated from an area called the heap. The heap is a dynamically growing and shrinking region of memory allocated for pointer variables.

MS-Pascal also provides two machine-oriented address types: one for addresses that can be represented in 16 bits, the other for addresses that require 32 bits.

Pointers are generally used for trees, graphs, and list processing. Use of pointers is portable, structured, and relatively safe.

Address types provide an interface to the hardware and operating system; their use is frequently unstructured, machine specific, low level, and unsafe. Both pointers and address types are discussed further in the following sections.

Pointer Types

A pointer type is a set of values that point to variables of a given type. The type of the variables pointed to is called the "reference type." Reference variables are all dynamically allocated from the heap with the NEW procedure. Pascal variables are normally allocated on the stack or at fixed locations.

You may perform only the following actions on pointers:

1. assign them
2. test them for equality and inequality with the two operators = and <>
3. pass them as value or reference parameters
4. dereference them with the up arrow (^)

Every pointer type includes the pointer value NIL. Pointers are frequently used to create list structures of records, as shown in the following example:

```
TYPE
  TREETIP = ^ TREE;
  TREE = RECORD
           VAL: INTEGER;
              {Value of TREE cell.}
           LEFT, RIGHT: TREETIP
              {Pointers to other TREETIP cells.}
              {Note recursive definition.}
         END;
```

Unlike most type declarations, the declaration for a pointer type can refer to a type of which it is itself a component. The declaration can also refer to a type declared later in the same TYPE section, as in TREE and TREETIP in the previous example.

Such a declaration is called a forward pointer declaration and permits recursive and mutually recursive structures. Because pointers are so often used in list structures, forward pointer declarations occur frequently.

The compiler checks for one ambiguous pointer declaration. Suppose the previous example was in a procedure nested in another procedure that also declared a type TREE. Then the reference type of TREETIP could be either the outer definition or the one following in the same TYPE section. MS-Pascal assumes the TREE type intended is the one later in the same TYPE section and gives the warning:

```
Pointer Type Assumed Forward
```

At the extend level, a pointer can have a super array type as a referent type. The actual upper bounds of the array are passed to the NEW procedure to create a heap variable of the correct size. Forward pointer declarations of the super array type are not allowed.

MS-Pascal conforms to the ISO requirement for strict compatibility between pointers. For example, you cannot declare two pointers with different types and then assign or compare them, even if they happen to point to the same underlying type. For example:

```
VAR PRA : ^ REAL;
    PRE : ^ REAL;
BEGIN PRA := PRE END; {This is illegal!}
```

Programs usually contain only one type declaration for a pointer to a given type. In the TREETIP example, the type of LEFT and RIGHT could be ^TREE instead of TREETIP, but then you couldn't assign variables of type TREETIP to these fields. However, it is sometimes useful to make sure that two classes of pointers are not used together, even if they point to the same type.

For example, suppose you have a type RESOURCE kept in a list and declare two types, OWNER and USER, of type ^RESOURCE. The compiler would catch assignment of OWNER values to USER variables and vice versa and issue a warning message.

In theory, pointers have nothing to do with actual machine addresses. In fact, a pointer may be implemented in different ways on different target machines. A pointer may be implemented as a normal address, as a segment offset address, as an offset from one or more fixed locations, or as an indirect address, among other possibilities.

If the initialization checking switch is on, a newly created pointer has an uninitialized value. If the NIL checking switch is on, pointer values are tested for various invalid values. Invalid values include NIL, uninitialized values, reference to a heap item that has been DISPOSEd, or a value that is not valid as a heap reference.

Address Types

As a system implementation language, MS-Pascal needs a method of creating, manipulating, and dereferencing actual machine addresses. The pointer type is only applicable to variables in the heap.

There are two kinds of addresses: relative and segmented. The keywords ADR and ADS refer to the relative address type and the segmented address type,respectively. As the following example shows, you use the keywords both as type clause prefixes and as prefix operators:

```
VAR INT_VAR : INTEGER;
    REAL_VAR : REAL;
    A_INT : ADR OF INTEGER;
    {Declaration of ADR variable}
    AS_REAL : ADS OF REAL;
    {Declaration of ADS variable}

BEGIN
  INT_VAR  := 1;
  {Normal integer variable}
  REAL_VAR := 3.1415;
  {Normal real variable}
  A_INT    := ADR INT_VAR;
  {ADR used as operator}
  AS_REAL  := ADS REAL_VAR;
  {ADS used as operator}
  WRITELN (A_INT^,AS_REAL^)
  {Note use of up arrow to dereference}
  {the address types.}
  {Output is 1 and 3.1415.}
END.
```

The characteristics of relative and segmented address types, as implemented for different machines, are shown in Table 8.1.

Table 8.1. Relative and Segmented Machine Addresses

Machine	ADR	ADS
8080	16-bit absolute	Same as ADR
8086	16-bit default data segment offset	16-bit offset, 16-bit segment
Z8000 (unsegmented)	16-bit data absolute	Same as ADR
Z8000 (segmented)	Same as ADS	16-bit segment, 16-bit offset

See your *Microsoft Pascal Compiler User's Guide* for details specific to your implementation of the compiler.

In MS-Pascal, you may declare a variable that is an address:

```
VAR X : ADR OF BYTE;
```

Then, with the following record notation, you can assign numeric values to the actual variable:

```
X.R := 16#FFFF
```

In an unsegmented environment, the .R (relative address) is the only record field available for ADR and ADS addresses.

Since MS-Pascal allows nondecimal numbering, you may specify the assigned value in hexadecimal notation. You may also assign to a segment field with the ADS type in a segmented environment, using the field notation .S (segment address). Thus, you may declare a variable of an ADS type and then assign values to its two fields:

```
VAR Y : ADS OF WORD;
      .
      .
Y.S := 16#0001
Y.R := 16#FFFF
```

As shown above, any 16-bit value can be directly assigned to address type variables, using the .R and .S fields. The ADR and ADS operators obtain these addresses directly. The example below assigns addresses this way to the variables X and Y:

```
VAR X : ADR OF BYTE;
    Y : ADS OF WORD;
    W : WORD;
    B : BYTE;
      .
      .
    X := ADR B;
    Y := ADS W;
```

MS-Pascal supports these two predeclared address types:

```
ADRMEM = ADR OF ARRAY [0..32766] OF BYTE;
ADSMEM = ADS OF ARRAY [0..32766] OF BYTE;
```

Since the type referred to by the address is an array of bytes, byte indexing is possible. For example, if A is of type ADRMEM, then A^[15] is the byte at the address A.R + 15, where .R specifies an actual 16-bit address.

You can use the address types for a constant address (a form of structured constant); you may also take the address of a constant or expression. For example:

```
TYPE ADRWORD = ADR OF WORD;
     ADSWORD = ADS OF WORD;
VAR W: WORD;
    R: ADRWORD;
CONST CONADR = ADRWORD (1234);
BEGIN
  W := CONADR^;
  {Get word at address 1234}
  W := ADSWORD (0, 32)^;
  {Get word at address 0:32}
  W := (ADS W).S;
  {Get value of DS segment register}
  R := ADR '123';
  {Get address of a constant value}
  R := ADR (W DIV 2 + 1);
  {Get address of expression value}
END;
```

However, constants or expressions that yield addresses cannot currently be used as the target of an assignment (or as a reference parameter or WITH record), as shown:

```
CONST ADSCON = ADSWORD (256, 64);   {OK}
FUNCTION SOME_ADDRESS: ADSWORD;     {OK}
BEGIN
  ADSWORD (0, 32)^ := W; {Not permitted}
  ADSCON^ := 12;         {Not permitted}
  SOME_ADDRESS^ := 100;  {Not permitted}
END;
```

Segment Parameters for the Address Types

Two keywords, VARS and CONSTS, are available as parameter prefixes, like VAR and CONST, to pass the segmented address of a variable. If P is of type ADSFOO, then P^ can be passed to a VARS formal parameter, such as VARS X: FOO, but cannot be passed to a VAR formal parameter.

In a segmented machine environment, a default data segment is assumed, in which case a VAR parameter is passed as the default data segment offset of a variable. A VARS parameter is passed as both the segment value and the offset value.

In the 8086 environment, both VARS parameters and ADS variables have the offset (.R) value in the WORD with the lower address and the segment (.S) value in the address plus two.

In the segmented Z8000 environment, the segment (.S) value is in the lower address and the offset (.R) value in the address plus two. Also, the ADR type is identical to the ADS type.

In the nonsegmented environment (e.g., 8080), VAR and CONST are identical to VARS and CONSTS. Since ADS and ADR are identical in a nonsegmented environment, the ADS type is useful in situations where the target environment may change. For example, in MS-Pascal, some primitive file system calls are declared with ADS parameters.

In pointer type declarations, the up arrow (^) prefixes the type pointed to; in program statements, it dereferences a pointer so that the value pointed to can be assigned or operated on. The up arrow also dereferences ADR and ADS types in program statements.

Component selection with the up arrow (^) is performed before the unary operators ADR or ADS. Because the up arrow (^) selector can appear after any address variable to produce a new variable, it can occur, for example, in the target of an assignment, a reference parameter, as well as in expressions. Since ADS and ADR are prefix operators, they are used only in expressions, where they apply only to a variable or constant or expression.

Pascal is a strongly typed language; two pointer variables are compatible only if they have the same type (it is not enough that they point to the same type). However, two address types are considered the same type if they are both ADR or both ADS types. This lets you assign an ADR OF WORD to an ADR OF STRING (200). Such an assignment would make it easy to wipe out part of memory by assigning a variable of type STRING (200) to the 200 bytes starting at the address of a WORD variable.

If P1 is type ADR OF STRING (200) and P2 is any ADR OF type, the assignment P1^ := P2^ generates fast code with no range checking. Although this capability is not safe, operating systems and other software sometimes require it.

ADR and ADS are not compatible with each other, but the .R notation should overcome or reduce the problem.

Using the Address Types

Within limits, you may combine and intermingle the two address types. The following example illustrates the rules that apply in a segmented environment:

```
VAR
  P: ADS OF DATA;
  {P is segmented address of type DATA.}
  Q: ADR OF DATA;
  {Q is relative address of type DATA.}
  X: DATA;
  {X is some variable of type DATA.}
BEGIN
  P := ADS X;
  {Assign the address of X to P.}
  X := P^;
  {Assign to X the value pointed to by P.}
  P := ADS P^;
  {Assign to P the address of the value whose}
  {address is pointed to by P. P is unchanged}
  {by this assignment.}
  Q := ADR X;
  {Assign the relative address of X to Q.}
  Q.R := (ADR X).R;
  {Assign the relative address of X to Q,}
  {using the WORD type.}
  P := ADS Q^;
  {Assign address of variable at Q to P.}
  {You can always apply ADS to ADR^.}
  Q := ADR P^;
  {Illegal; you cannot apply ADR to ADS ^.}
  P.R := 16#8000;
  {Assign 32768 to P's offset field.}
  P.S := 16;
  {Assign 16 to P's segment field.}
  Q.R := P.R + 4;
  {Assign P's offset plus 4 to be the value of Q.}
END;
```

See also the examples given in Chapter 8, "Address Types."

Notes on Reference Types

The address type and pointer type should be treated as two distinct types. The pointer type, in theory, is just an undefined mapping from a variable to another variable. The method of implementation is undefined. However, the address type deals with actual machine addresses.

Therefore, the pointer type is an abstract data type that works the same in all implementations; the address type is generally not portable, unless used with some caution. Address types are portable only if you restrict yourself to using ADS and never assign to fields. Even with these restrictions, however, they can be quite useful.

The following special facilities that use pointer variables are not allowed with address variables.

1. The NEW and DISPOSE procedures are only permitted with pointers. NIL does not apply to the address type. There are no special address values for empty, uninitialized, or invalid addresses.

2. The type "address of super array type" is not supported in the same way as "pointer to super array type." Getting the address of a super array variable is still permitted with ADR and ADS. For example, if a procedure or function formal parameter is declared as VAR S: STRING, then within the procedure or function, the expression ADS S is fine. Unlike a pointer, the address does not contain any upper bounds.

PACKED Types

Any of the structured types can be PACKED. This could economize storage at the possible expense of access time or access code space. However, in MS-Pascal, some limitations on the use of PACKED structures currently apply:

1. The prefix PACKED is always ignored, except for type checking, in sets, files, and arrays of characters, and in most versions of MS-Pascal has no actual effect on the representation of records and other arrays. Furthermore, PACKED can only precede one of the structure names ARRAY, RECORD, SET, or FILE; it cannot precede a type identifier. For example, if COLORMAP is the identifier for an unpacked array type, "PACKED COLORMAP" is not accepted.

2. A component of a PACKED structure cannot be passed as a reference parameter or used as the record of a WITH statement, unless the structure is of a string type. Also, obtaining the address of a PACKED component with ADR or ADS is not permitted.

3. A PACKED prefix only applies to the structure being defined: any components of that structure that are also structures are not packed unless you explicitly include the reserved word PACKED in their definition. The only exception to this rule, n-dimensional arrays, is discussed in Chapter 6, "Arrays."

Procedural and Functional Types

Procedural and functional types are different from other MS-Pascal types. (Wherever the term "procedural" is used from here on, both procedural and functional is implied.) You may not declare an identifier for a procedural type in a TYPE section; nor may you declare a variable of a procedural type. However, you may use procedural types to declare the type of a procedural parameter, and in this sense they conform to the Pascal idea of a type.

A procedural type defines a procedure or function heading and gives any parameters. For a function, it also defines the result type. The syntax of a procedural type is the same as a procedure or function heading, including any attributes. There are no procedural variables in MS-Pascal, only procedural parameters.

Example of a procedural type declaration:

```
PROCEDURE ZERO (FUNCTION FUN (X, Y: REAL): REAL)
```

The parameter identifiers in a procedural type (X and Y in the previous example) are ignored; only their type is important.

See Chapter 13, "Procedural and Functional Parameters," for more information about procedural types in MS-Pascal.

Chapter 9

CONSTANTS

What Is a Constant?

A constant is a value that is known before a program starts and that will not change as the program progresses. Examples of constants include the number of days in the week, your birth date, the name of your dog, or the phases of the moon.

A constant may be given an identifier, but you cannot alter the value associated with that identifier during the execution of the program. When you declare a constant, its identifier becomes a synonym for the constant itself.

Each constant implicitly belongs to some category of data:

1. Numeric constants (discussed in Chapter 9, "Numeric Constants") are one of the several number types: REAL, INTEGER, WORD, or INTEGER4.
2. Character constants (discussed in Chapter 9, "Character Strings") are strings of characters enclosed in single quotation marks and are called "string literals" in MS-Pascal.
3. Available at the extend level, structured constants (discussed in Chapter 9, "Structured Constants") include constant arrays, records, and typed sets.

Also available at the extend level, constant expressions (discussed in Chapter 9, "Constant Expressions") let you compute a constant based on the values of previously declared constants in expressions.

The identifiers defined in an enumerated type are constants of that type and cannot be used directly with numeric (or string) constant expressions. These identifiers can be used with the ORD, WRD, or CHR functions (e.g., ORD (BLUE)). The extend level also permits directly reading and writing the enumerated type's constant identifiers as character strings.

TRUE and FALSE are predeclared constants of type BOOLEAN and can be redeclared. NIL is a constant of any pointer type; however, because it is a reserved word, you may not redefine it. Also, the null set is a constant of any set type.

Numeric statement labels have nothing to do with numeric constants; you may not use a constant identifier or expression as a label. Internally, all constants are limited in length to a maximum of 255 bytes.

Declaring Constant Identifiers

Declaring a constant identifier introduces the identifier as a synonym for the constant. You put these declarations in the CONST section of a compiland, procedure, or function.

The general form of a constant identifier declaration is the identifier followed by an equal sign and the constant value. The following program fragment includes three statements that identify constants (beginning after the word "CONST"):

```
PROGRAM DEMO (INPUT, OUTPUT);
CONST DAYSINYEAR = 365;
      DAYSINWEEK = 7;
      NAMEOFPLANET = 'EARTH';
```

In this example, the numbers 365 and 7 are numeric constants; 'EARTH' is a string literal constant and must be enclosed in single quotation marks.

When you compile a program, the constant identifiers are not actually defined until after the declarations are processed. Thus, a constant declaration like the following has no meaning:

```
N = -N
```

The ISO standard defines a strict order in which to set out the declarations in the declaration section of a program:

```
CONST MAX = 10;
TYPE NAME = PACKED ARRAY [1..MAX] OF CHAR;
VAR FIRST : NAME;
```

MS-Pascal relaxes this order and, in fact, allows more than one instance of each kind of declaration:

```
TYPE COMPLEX = RECORD R, I : REAL END;
CONST PII = COMPLEX (3.1416, 00);
VAR PIX : COMPLEX;
TYPE IVEC = ARRAY [1..3] OF COMPLEX;
CONST PIVEC = IVEC (PII, PII, COMPLEX (0.0, 1.0));
```

Numeric Constants

Numeric constants are irreducible numbers such as 45, 12.3, and 9E12. The notation of a numeric constant generally indicates its type: REAL, INTEGER, WORD, or INTEGER4.

Numbers can have a leading plus sign (+) or minus sign (-), except when the numbers are within expressions. Therefore:

```
ALPHA := +10     {Is legal}
ALPHA + -10      {Is illegal}
```

Blanks embedded within constants are not permitted.

The compiler truncates any number that exceeds a certain maximum number of characters and gives a warning when this occurs. The maximum length of constants (either 19 or 31) is the same as the maximum length of identifiers. For the maximum length of constants and identifiers in a particular version of the language, see Appendix A, "Version Specifics," in your *Microsoft Pascal Compiler User's Guide.*

The syntax for numeric constants applies not only to the actual text of programs, but also to the content of textfiles read by a program.

Examples of numeric constants:

```
 123              0.17
+12.345           007
-1.7E-10         -26.0
 17E+3            26.0E12
 -17E3            1E1
```

Numeric constants can appear in any of the following:

1. CONST sections
2. expressions
3. type clauses
4. set constants
5. structured constants
6. CASE statement CASE constants
7. variant record tag values

The different types of numeric constants are discussed in detail in the following sections.

REAL Constants

The type of a number is REAL if the number includes a decimal point or exponent. The REAL value range depends on the REAL number unit of the target machine. Generally, either the IEEE or the Microsoft REAL number format is used. This provides about seven digits of precision, with a maximum value of about 1.701411E38.

There is, however, a distinction between REAL values and REAL constants. The REAL constant range may be a subset of the REAL value range. In Microsoft format, REAL numeric constants must be greater than or equal to 1.0E-38 and less than 1.0E+38. In IEEE format, REAL numeric constants are kept in double precision and so can range from about 1E-306 to 1E306.

The compiler issues a warning if there is not at least one digit on each side of a decimal point. A REAL number starting or ending with a decimal point may be misleading. For example, because left parenthesis-period substitutes for left square bracket, and right parenthesis-period for right square bracket, the following:

```
(.1+2.)
```

is interpreted as:

```
[1+2]
```

Scientific notation in REAL numbers (as in 1.23E-6 or 4E7) is supported. The decimal point and exponent sign are optional when an exponent is given. Both the uppercase "E" and the lowercase "e" are allowed in REAL numbers. "D" and "d" are also allowed to indicate an exponent. This provides compatibility with other languages.

When IEEE REAL4 and REAL8 format are used, all real constants are stored in REAL8 (double precision) format. If you require a single precision REAL4 constant, declare a REAL4 variable and give it your real constant value in a VALUE section. (You may wish to give this variable the READONLY attribute as well.)

Versions of the compiler that run on one machine but generate code for another may lose a small amount of significance in REAL constants.

INTEGER, WORD, and INTEGER4 Constants

The type of a non-REAL numeric constant is INTEGER, WORD, or INTEGER4. Table 9.1 shows the range of values that constants of each of these types can assume.

Table 9.1. INTEGER, WORD, and INTEGER4 Constants

Type	Range of Values (minimum / maximum)	Predeclared Constant
INTEGER	-MAXINT to MAXINT	MAXINT=32767
WORD	0 to MAXWORD	MAXWORD=65535
INTEGER4	-MAXINT4 to MAXINT4	MAXINT4=2147483647

MAXINT, MAXWORD, and MAXINT4 are all predeclared constant identifiers.

One of three things happens when you declare a numeric constant identifier:

1. A constant identifier from -MAXINT to MAXINT becomes an INTEGER.
2. A constant identifier from MAXINT+1 to MAXWORD becomes a WORD.
3. A constant identifier from -MAXINT4 to -MAXINT-1 or MAXWORD+1 to MAXINT4 becomes an INTEGER4.

However, any INTEGER type constant (including constant expressions and values from -32767 to -1) automatically changes to type WORD if necessary; if the INTEGER value is negative, 65536 is added to it and the underlying 16-bit value is not changed.

For example, you can declare a subrange of type WORD as WRD(0)..127; the upper bound of 127 is automatically given the type WORD. The reverse is not true; constants of type WORD are not automatically changed to type INTEGER.

The ORD and WRD functions also change the type of an ordinal constant to INTEGER or WORD. Also, any INTEGER or WORD constant automatically changes to type INTEGER4 if necessary, but the reverse is not true.

Examples of relevant conversions are given in Table 9.2.

Table 9.2. Constant Conversions

Constant	Assumed Type
0	INTEGER could become WORD or INTEGER4
-32768	INTEGER4 only
32768	WORD could become INTEGER4
0..20000	INTEGER subrange
0..50000	WORD subrange
0..80000	Invalid: no INTEGER4 subranges
-1..50000	Invalid: becomes 65535..50000 (i.e., -1 is treated as 65536)

At the standard level, any numeric constant (i.e., literal or identifier) may have a plus (+) or minus (-) sign.

Nondecimal Numbering

At the extend level, MS-Pascal supports not only decimal number notation, but also numbers in hexadecimal, octal, binary, or other base numbering (where the base can range from 2 to 36). The number sign (#) acts as a radix separator.

Examples of numbers in nondecimal notation:

```
16#FF02
10#987
8#776
2#111100
```

Leading zeros are recognized in the radix, so a number like 008#147 is permitted. In hexadecimal notation, upper or lowercase letters A through F are permitted. A nondecimal constant without the radix (such as #44) is assumed to be hexadecimal. Nondecimal notation does not imply a WORD constant and may be used for INTEGER, WORD, or INTEGER4 constants. You must not use nondecimal notation for REAL onstants or numeric statement labels.

Character Strings

Most Pascal manuals refer to sequences of characters enclosed in single quotation marks as "strings." In MS-Pascal, they are called "string literals" to distinguish them from string constants, which may be expressions, or values of the STRING type.

A string constant contains from 1 to 255 characters. A string constant longer than one character is of type PACKED ARRAY [1..n] OF CHAR, also known in MS-Pascal as the type STRING (n). A string constant that contains just one character is of type CHAR. However, the type changes from CHAR to PACKED ARRAY [1..1] OF CHAR (e.g., STRING (1)) if necessary. For example, a constant ('A') of type CHAR could be assigned to a variable of type STRING (1).

A literal apostrophe (single quotation mark) is represented by two adjacent single quotation marks (e.g., 'DON''T GO'). The null string ('') is not permitted. A string literal must fit on a line. The compiler recognizes string literals enclosed in double quotations marks (") or accent marks (`), instead of single quotation marks, but issues a warning message when it encounters them.

The constant expression feature (discussed in Chapter 9, "Constant Expressions") permits string constants made up of concatenations of other string constants, including string constant identifiers, the CHR () function, and structured constants of type STRING. This is useful for representing string constants that are longer than a line or that contain nonprinting characters. For example:

```
'THIS IS UNDERLINED' * CHR(13) * STRING (DO 18 OF '_')
```

The LSTRING feature of MS-Pascal adds the super array type LSTRING. LSTRING is similar to PACKED ARRAY [0..n] OF CHAR, except that element 0 contains the length of the string, which can vary from 0 to a maximum of 255. (See Chapter 6, "LSTRINGs," for a discussion of LSTRINGs.) For now, note that, if necessary, a constant of type STRING (n) or CHAR changes automatically to type LSTRING.

NULL is a predeclared constant for the null LSTRING, with the element 0 (the only element) equal to CHR (0). NULL cannot be concatenated, since it is not of type STRING. It is the only constant of type LSTRING.

Examples of string literal declarations:

```
NAME = 'John Jacob';        {a legal string literal}
LETTER = 'Z';               {LETTER is of type CHAR}
QUOTED_QUOTE = '''';        {Quotes quote}
NULL_STRING = NULL;         {legal}
NULL_STRING = '';           {illegal}
DOUBLE = "OK";              {generates a warning}
```

Structured Constants

Standard Pascal permits only the numeric and string constants already mentioned, the pointer constant value NIL, and untyped constant sets.

At the extend level of MS-Pascal, however, you may use constant arrays, records, and typed sets. Structured constants can be used anywhere a structured value is allowed, in expressions as well as in CONST and VALUE sections.

1. An array constant consists of a type identifier followed by a list of constant values in parentheses separated by commas.

   Example of an array constant:

   ```
   TYPE   VECT_TYPE = ARRAY [-2..2] OF INTEGER;
   CONST  VECT = VECT_TYPE (5, 4, 3, 2, 1);
   VAR    A : VECT_TYPE;
   VALUE  A := VECT;
   ```

2. A record constant consists of a type identifier followed by a list of constant values in parentheses separated by commas.

   Example of a record constant:

   ```
   TYPE REC_TYPE = RECORD
          A, B: BYTE;
          C, D: CHAR;
        END;
   CONST RECR = REC_TYPE (#20, 0, 'A', CHR (20));
   VAR FOO : REC_TYPE;
   VALUE FOO := RECR;
   ```

3. A set constant consists of an optional set type identifier followed by set constant elements in square brackets. Set constant elements are separated by commas. A set constant element is either an ordinal constant, or two ordinal constants separated by two dots to indicate a range of constant values.

   Example of a set constant:

```
TYPE COLOR_TYPE = SET OF
   (RED, BLUE, WHITE, GREY, GOLD);
CONST SETC = COLOR_TYPE [RED, WHITE .. GOLD];
VAR RAINBOW : COLOR_TYPE;
VALUE RAINBOW := SETC;
```

A constant within a structured array or record constant must have a type that can be assigned to the corresponding component type. For records with variants, the value of a constant element corresponding to a tag field selects a variant, even if the tag field is empty. The number of constant elements must equal the number of components in the structure, except for super array type structured constants. Nested structured constants are permitted.

An array or record constant nested within another structured constant must still have the preceding type identifier. For this reason, a super array constant can have only one dimension (see Chapter 6, "Super Arrays," for a discussion of super arrays). The size of the representation of a structured constant must be from 1 to 255 bytes. If this 255-byte limit is a problem, declare a structured variable with the READONLY attribute, and initialize its components in a VALUE section.

Example of a complex structured constant:

```
TYPE R3 = ARRAY [1..3] OF REAL;
TYPE SAMPLE = RECORD I: INTEGER;
                     A: R3;
                     CASE BOOLEAN OF
                     TRUE: (S: SET OF 'A'..'Z';
                            P: ^ SAMPLE);
                     FALSE:(X: INTEGER);
              END;
CONST SAMP_CONST= SAMPLE (27, R3 (1.4, 1.4, 1.4),
              TRUE, ['A','E','I'], NIL);
```

Constant elements can be repeated with the phrase DO <n> OF <constant>, so the previous example could have included "DO 3 OF 1.4" instead of "1.4, 1.4, 1.4".

MS-Pascal does not support set constant expressions, such as ['_'] + LETTERS, or file constant expressions. The constant 'ABC' of type STRING (3) is equivalent to the structured constant STRING ('A', 'B', 'C'). LSTRING structured constants are not permitted; use the corresponding STRING constants instead.

Structured constants (and other structured values, such as variables and values returned from functions) can be passed by reference using CONST parameters. For more information, see Chapter 13, "Procedure and Function Parameters."

There are two kinds of set constants: one with an explicit type, as in CHARSET ['A'..'Z'], and one with an unknown type, as in [20..40]. You may use either in an expression or to define the value of a constant identifier. Set constants with an explicit type may also be passed as a reference (CONST) parameter. Sets of unknown type are unpacked, but the type changes to PACKED if necessary. Passing sets by reference is generally more efficient than passing them as value parameters.

Constant Expressions

Constant expressions in MS-Pascal allow you to compute constants based on the values of previously declared constants in expressions. Constant expressions can also occur within program statements.

Example of a constant expression declaration:

```
CONST HEIGHT_OF_LADDER = 6;
      HEIGHT_OF_MAN    = 6;
      REACH = HEIGHT_OF_LADDER + HEIGHT_OF_MAN;
```

Because a constant expression may contain only constants that you have declared earlier, the following is illegal:

```
CONST MAX = A + B;
      A = 10;
      B = 20;
```

Certain functions may be used within constant expressions. For example:

```
CONST A = LOBYTE (-23) DIV 23;
      B = HIBYTE (-A);
```

Table 9.3 shows the functions and operators you may use with REAL, INTEGER, WORD, and other ordinal constants, such as enumerated and subrange constants.

Table 9.3. Constant Operators and Functions

Type of Operand	Functions and Operators
REAL, INTEGER	Unary plus (+) Unary minus (-)
INTEGER, WORD	+ DIV OR HIBYTE() - MOD NOT LOBYTE() * AND XOR BYWORD()
Ordinal types	< <= CHR() LOWER() > >= ORD() UPPER() = <> WRD()
Boolean	AND NOT OR
ARRAY	LOWER() UPPER()
Any type	SIZEOF() RETYPE()

Examples of constant expressions:

```
CONST FOO = (100 + ORD('X')) * 8#100 + ORD('Y');
      MAXSIZE = 80;
      X = (MAXSIZE > 80) OR (IN_TYPE = PAPERTAPE);
      {X is a BOOLEAN constant}
```

In addition to the operators shown in Table 9.3 for numeric constants, you may use the string concatenation operator (*) with string constants, as follows:

```
CONST A = 'abcdef';
      M = CHR (109); {CHR is allowed}
      ATOM = A * 'ghijkl' * M;
      {ATOM = 'abcdefghijklm'}
```

These constants can span more than one line, but are still limited to the 255 character maximum. These string constant expressions are allowed wherever a string literal is allowed, except in metacommands.

Chapter 10

VARIABLES AND VALUES

What Is a Variable?

A variable is a value that is expected to change during the course of a program. Every variable must be of a specific data type. A variable may have an identifier.

If A is a variable of type INTEGER, then the use of A in a program actually refers to the data denoted by A. For example:

```
VAR A: INTEGER;
    BEGIN
    A := 1;
    A := A + 1;
END;
```

These statements would first assign a value of 1 to the data denoted by A, and subsequently assign it a value of 2.

Variables are manipulated by using some sort of notation to denote the variable, in the simplest case, a variable identifier. In other cases, variables may be denoted by array indices or record fields or the dereferencing of pointer or address variables.

The compiler itself may sometimes create "hidden" variables, allocated on the stack, in circumstances like the following:

1. When you call a function that will return a structured result, the compiler allocates a variable in the caller for the result.
2. When you need the address of an expression (e.g., to pass it as a reference parameter or to use it as a WITH statement record or with ADR or ADS), the compiler allocates a variable for the value of the expression.

3. The initial and final values of a FOR loop may require allocating a variable.
4. When the compiler evaluates an expression, it may allocate a variable to store intermediate results.
5. Every WITH statement requires a variable to be allocated for the address of the WITH's record.

Declaring a Variable

A variable declaration consists of the identifier for the new variable, followed by a colon and a type. You may declare variables of the same type by giving a list of the variable identifiers, followed by their common type. For example:

```
VAR XCOORD, YCOORD: REAL
```

You may declare a variable in any of the following locations:

1. VAR section of a program, procedure, function, module, interface, or implementation
2. formal parameter list of a procedure, function, or procedural parameter

In a VAR section, you may declare a variable to be of any legal type; in a formal parameter list, you may include only a type identifier (i.e., you may not declare a type in the heading of a procedure or function). For example:

```
PROCEDURE NAME (GEORGE: ARRAY [1..10] OF COLOR)
{Illegal; GEORGE is of a new type.}

VAR VECTOR_A: VECTOR (10)
{Legal; VECTOR (10) is a type derived from}
{a super type.}
```

Each declaration of a file variable F of type FILE OF T implies the declaration of a buffer variable of type T, denoted by F^. At the extend level, a file declaration also implies the declaration of a record variable of type FCBFQQ, whose fields are denoted as F.TRAP, F.ERRS, F.MODE, and so on. See Chapter 7, "The Buffer Variable," and Chapter 7, "Extend Level I/O," for further information on buffer variables and FCBFQQ fields, respectively.

The VALUE Section

The VALUE section in MS-Pascal lets you give initial values to variables in a program, module, procedure, or function. You may also initialize the variable in an implementation, but not in an interface.

The VALUE section may include only statically allocated variables, that is, any variable declared at the program, module, or implementation level, or a variable with the STATIC or PUBLIC attribute. Variables with the EXTERN or ORIGIN attribute cannot occur in a VALUE section, since they are not allocated by the compiler.

The VALUE section may contain assignments of constants to entire variables or to components of variables. For example:

```
VAR ALPHA : REAL;
    ID : STRING (7);
    I : INTEGER;

VALUE ALPHA := 2.23;
ID[1] := 'J';
I := 1;
```

However, within a VALUE section, you may not assign a variable to another variable. The last line in the following example is illegal, since "I" must be a constant:

```
CONTS MAX = 10;
VAR I, J : INTEGER;
VALUE I := MAX;
      J := I;
```

If the $ROM metacommand is off, variables are initialized by loading the static data segment. If the $ROM metacommand is on, the VALUE section generates an error message since ROM-based systems usually cannot statically initialize data.

Using Variables and Values

At the standard level of MS-Pascal, denotation of a variable may designate one of three things:

1. an entire variable
2. a component of a variable
3. a variable referenced by a pointer

A value may be any of the following:

1. a variable
2. a constant
3. a function designator
4. a component of a value
5. a variable referenced by a reference value

At the extend level, a function can also return an array, record, or set. The same syntax used for variables may be used to denote components of the structures these functions return.

This feature also allows you to dereference a reference type that is returned by a function. However, you may only use the function designator as a value, not as a variable. For example, the following is illegal:

```
F (X, Y)^ := 42;
```

Also at the extend level, you may declare constants of a structured type. Components of a structured constant use the same syntax as variables of the same type (see Chapter 9, "Constant Expressions," for further discussion of this topic).

Examples of structured constant components:

```
TYPE REAL3 = ARRAY [1..3] OF REAL;
{an array type}
CONST PIES = REAL3 (3.14, 6.28, 9.42);
{an array constant}
  .
  .
  X := PIES [1] * PIES [3];
  {i.e., 3.14 * 9.42}
  Y := REAL3 (1.1, 2.2, 3.3) [2];
  {i.e., 2.2}
```

Components of Entire Variables and Values

At the standard level, a variable identifier denotes an entire variable. A variable, function designator, or constant denotes an entire value.

A component of a variable or value is denoted by the identifier followed by a selector that specifies the component. The form of a selector depends on the type of structure (array, record, file, or reference).

Indexed Variables and Values

A component of an array is denoted by the array variable or value, followed by an index expression. The index expression must be assignment compatible with the index type in the array type declaration. An index type must always be an ordinal type. The index itself must be enclosed in brackets following the array identifier.

Examples of indexed variables and values:

```
ARRAY_OF_CHAR ['C']
{Denotes the Cth element.}

'STRING CONSTANT' [6]
{Denotes the 6th element, the letter 'G'.}

BETAMAX [12] [-3]
BETAMAX [12,-3]
{These two say the same thing.}

ARRAY_FUNCTION (A, B) [C, D]
{Denotes a component of a two-dimensional array}
{returned by ARRAY_FUNCTION (A, B). A and B are}
{actual parameters.}
```

You may specify the current length of an LSTRING variable, LSTR, in either of two ways:

1. with the notation LSTR [0], to access the length as a CHAR component
2. with the notation LSTR.LEN, to access the length as a BYTE value

Field Variables and Values

A component of a record is denoted by the record variable or value followed by the field identifier for the component. Fields are separated by the period (.). In a WITH statement, you give the record variable or value only once. Within the WITH statement, you may use the field identifier of a record variable directly.

Examples of field variables and values:

```
PERSON.NAME := 'PETE'

PEOPLE.DRIVERS.NAME := 'JOAN'

WITH PEOPLE.DRIVERS DO NAME := 'GERI'

RECURSING-FUNC ('XYZ').BETA
{Selects BETA field of record returned}
{by the function named RECURSIVE_FUNC.}

COMPLEX_TYPE (1.2, 3.14).REAL_PART
```

Record field notation also applies to files for FCBFQQ fields, to address type values for numeric representations, and to LSTRINGs for the current length.

File Buffers and Fields

At any time, only one component of a file is accessible. The accessible component is determined by the current file position and represented by the buffer variable. Depending on the status of the buffer variable, fetching its value may first read the value from the file. (This is called "lazy evaluation"; see Chapter 15, "Lazy Evaluation," for details.)

If a file buffer variable is passed as a reference parameter or used as a record of a WITH statement, the compiler issues a warning to alert you to the fact that the value of the buffer variable may not be correct after the position of the file is changed with a GET or PUT procedure.

Examples of file reference variables:

```
INPUT^
ACCOUNTS_PAYABLE.FILE^
```

Reference Variables

Reference variables or values denote data that refers to some data type. There are three kinds of reference variables and values:

1. pointer variables and values
2. ADR variables and values
3. ADS variables and values

In general, a reference variable or value "points" to a data object. Thus, the value of a reference variable or value is a reference to that data object. To obtain the actual data object pointed to, you must "dereference" the reference variable by appending an up arrow (^) to the variable or value.

Example using pointer values:

```
VAR P, Q : ^INTEGER;
{P and Q are pointers to integers.}

NEW (P); NEW (Q);
{P and Q are assigned reference values to}
{regions in memory corresponding to data}
{objects of type INTEGER.}

P := Q;
{P and Q now point to the same region}
{in memory.}

P^ := 123;
{Assigns the value 123 to the INTEGER value}
{pointed to by P. Since Q points to this}
{location as well, Q^ is also assigned 123.}
```

Using NIL^ is an error (since a NIL pointer does not reference anything). At the extend level, you may also append an up arrow (^) to a function designator for a function that returns a pointer or address type. In this case, the up arrow denotes the value referenced by the return value. This variable cannot be assigned to or passed as a reference parameter.

Examples of functions returning reference values:

```
DATA1 := FUNK1 (I, J)^
{FUNK1 returns a reference value. The up arrow}
{dereferences the reference value returned,}
{assigning the referenced data to DATA1.}

DATA2 := FUNK2 (K, L)^.FOO [2]
{FUNK2 returns a reference value. The up arrow}
{dereferences the reference value returned. In}
{this case, the dereferenced value is a record.}
{The array component FOO [2] of that record is}
{assigned to the variable DATA2.}
```

If P is of type ADR OF some type, then P.R denotes the address value of type WORD. If P is of type ADS OF some type, then P.R denotes the offset portion of the address and P.S denotes the segment portion of the address. Both portions are of type WORD.

Examples of address variables:

```
BUFF_ADR.R
DATA_AREA.S
```

Attributes

At the extend level of MS-Pascal, a variable declaration or the heading of a procedure or function may include one or more attributes. A variable attribute gives special information about the variable to the compiler.

Table 10.1 displays the attributes provided by MS-Pascal for variables.

Table 10.1. Attributes for Variables

Attribute	**Variable**
STATIC	Allocated at a fixed location, not on the stack.
PUBLIC	Accessible by other modules with EXTERN, implies STATIC.
EXTERN	Declared PUBLIC in another module, implies STATIC.
ORIGIN	Located at specified address, implies STATIC.
PORT	I/O address, implies STATIC.
READONLY	Cannot be altered or written to.

The EXTERN attribute is also a procedure and function directive; PUBLIC and ORIGIN are also procedure and function attributes. See Chapter 13, "Attributes and Directives," for a discussion of procedure and function attributes and directives. Chapter 10 discusses each of the variable attributes in detail.

You may only give attributes for variables in a VAR section. Specifying variable attributes in a TYPE section or a procedure or function parameter list is not permitted.

You give one or more attributes in the variable declaration, enclosed in brackets and separated by commas (if specifying more than one attribute).

The brackets may occur in either of two places:

1. An attribute in brackets after a variable identifier in a VAR section applies to that variable only.
2. An attribute in brackets after the reserved word VAR applies to all of the variables in the section.

Examples that specify variable attributes:

```
VAR A, B, C [EXTERN] : INTEGER;
{Applies to C only.}

VAR [PUBLIC] A, B, C : INTEGER;
{Applies to A, B, and C.}

VAR [PUBLIC] A, B, C [ORIGIN 16#1000] : INTEGER;
{A, B, and C are all PUBLIC. ORIGIN of C}
{is the absolute hexadecimal address 1000.}
```

The STATIC Attribute

The STATIC attribute gives a variable a unique, fixed location in memory. This is in contrast to a procedure or function variable that is allocated on the stack or one that is dynamically allocated on the heap. Use of STATIC variables can save time and code space, but increases data space.

All variables at the program, module, or unit level are automatically assigned a fixed memory location and given the STATIC attribute.

Functions and procedures that use STATIC variables can execute recursively, but STATIC variables must be used only for data common to all invocations. Since most of the other variable attributes imply the STATIC attribute, the trade-off between savings in time and code space or reduced data space applies to the PUBLIC, EXTERN, ORIGIN, and PORT attributes as well.

Files declared in a procedure or function with the STATIC attribute are initialized when the routine is entered; they are closed when the routine terminates like other files. However, other STATIC variables are only initialized before program execution. This means that, except for open FILE variables, STATIC variables can be used to retain values between invocations of a procedure or function.

Examples of STATIC variable declarations:

```
VAR VECTOR [STATIC]: ARRAY [0..MAXVEC] OF INTEGER;
VAR [STATIC] I, J, K: 0..MAXVEC;
```

The STATIC attribute does not apply to procedures or functions, as some other attributes do.

The PUBLIC and EXTERN Attributes

The PUBLIC attribute indicates a variable that may be accessed by other loaded modules; the EXTERN attribute identifies a variable that resides in some other loaded module. The identifier is passed to the target linker in the generated code object file (where it may be truncated if the linker imposes a length restriction).

Variables given the PUBLIC or EXTERN attribute are implicitly STATIC.

Examples of PUBLIC and EXTERN variable declarations:

```
VAR [EXTERN] GLOBE1, GLOBE2: INTEGER;
{The variables GLOBE1 and GLOBE2 are declared}
{EXTERN, meaning that they must be declared}
{PUBLIC in some other loaded module.}

VAR BASE_PAGE [PUBLIC, ORIGIN #12FE]: BYTE;
{The variable BASE_PAGE is located at 12FE,}
{hexadecimal. Because it is also PUBLIC, it can}
{be accessed from other loaded modules that}
{declare BASE_PAGE with the EXTERN attribute.}
```

PUBLIC variables are usually allocated by the compiler, unless you also give them an ORIGIN. Giving a variable both the PUBLIC and ORIGIN attributes tells the loader that a global name has an absolute address. PUBLIC cannot be combined with PORT.

If both PUBLIC and ORIGIN are present, the compiler does not need the loader to resolve the address. However, the identifier is still passed to the linker for use by other modules.

EXTERN variables are not allocated by the compiler. Nor do they have an ORIGIN, since giving both EXTERN and ORIGIN implies two different ways to access the variable. The reserved word EXTERNAL is synonymous with EXTERN. This increases portability from other Pascals, since others commonly use one of the two.

Variables in the interface of a unit are automatically given either the PUBLIC or EXTERN attribute. If a program, module, or unit USES an interface, its variables are made EXTERN; if you compile the IMPLEMENTATION of the interface, its variables are made PUBLIC.

The ORIGIN and PORT Attributes

The ORIGIN attribute directs the compiler to locate a variable at a given memory address; the PORT attribute specifies some kind of I/O address. In either case, the address must be a constant of any ordinal type. I/O ports, interrupt vectors, operating system data, and other related data can be accessed with ORIGIN or PORT variables.

Examples of ORIGIN and STATIC variable declarations:

```
VAR KEYBOARDP [PORT 16#FFF2]: CHAR;
VAR INTRVECT [ORIGIN 8#200]: WORD;
```

Variables with ORIGIN or PORT attributes are implicitly STATIC. Also, they inhibit common subexpression optimization. For example, if GATE has the ORIGIN attribute, the two statements X := GATE; Y := GATE access GATE twice in the order given, instead of using the first value for both assignments. This ensures correct operation if GATE is a memory-mapped input port. However, if GATE is passed as a reference parameter, references to the parameter may be optimized away. For this reason, PORT variables cannot be passed as reference parameters.

ORIGIN and PORT variables are never allocated or initialized by the compiler. The associated address only indicates where the variable is found. ORIGIN always implies a memory address, but the meaning of PORT varies with the implementation.

In most implementations, I/O is assumed to be memory mapped, so PORT is just a synonym for ORIGIN. Other implementations use the machine's native input and output instructions. Still others call port input and output routines for every access.

For more information on the PORT attribute, see Appendix A, "Version Specifics," in your *Microsoft Pascal Compiler User's Guide.*

Giving the PORT and ORIGIN attributes in brackets immediately following the VAR keyword is ambiguous and generates an error during compilation. (It would be unclear to the compiler whether all variables following should be at the same address or whether addresses should be assigned sequentially.)

```
VAR [ORIGIN 0] FIRST, SECOND:  BYTE;  {ILLEGAL!}
```

ORIGIN (but not PORT) permits a segmented address using "segment: offset" notation.

```
VAR SEGVECT [ORIGIN 16#0001:16#FFFE]:  WORD;
```

Currently, a variable with a segmented ORIGIN cannot be used as the control variable in a FOR statement.

The READONLY Attribute

The READONLY attribute prevents assignments to a variable. It also prevents the variable being passed as a VAR or VARS parameter. Also, a READONLY variable cannot be read with a READ statement or used as a FOR control variable. You may use READONLY with any of the other attributes.

Examples of READONLY variable declarations:

```
VAR INPORT [PORT 12, READONLY]: BYTE;
{INPORT is a READONLY PORT variable.}

VAR [READONLY] I, J [PUBLIC], K [EXTERN]: INTEGER;
{I, J, and K are all READONLY;}
{J is also PUBLIC; K is also EXTERN.}
```

CONST and CONSTS parameters, as well as FOR loop control variables (while in the body of the loop), are automatically given the READONLY attribute. READONLY is the only variable attribute that does not imply STATIC allocation.

A variable that is both READONLY and either PUBLIC or EXTERN in one source file is not necessarily READONLY when used in another source file. The READONLY attribute does not apply to procedures and functions.

Combining Attributes

You may give a variable multiple attributes. Separate the attributes with commas and enclose the list in brackets, as shown:

```
VAR [STATIC]
X, Y, Z [ORIGIN #FFFE, READONLY]: INTEGER;
```

In this example, Z is a STATIC, READONLY variable with an ORIGIN at hexadecimal FFFE. These rules apply when you are combining attributes:

1. If you give a variable the EXTERN attribute, you may not give it the PORT, ORIGIN, or PUBLIC attributes in the current compiland.
2. If you give a variable the PORT attribute, you may not give it the ORIGIN, PUBLIC, or EXTERN attributes at all.
3. If you give a variable the ORIGIN attribute, you may not also give it the PORT or EXTERN attributes. However, you may combine ORIGIN with PUBLIC.
4. If you give a variable the PUBLIC attribute, you may not also give it the PORT or EXTERN attributes. However, you may combine PUBLIC with ORIGIN.
5. You may use STATIC and READONLY with any other attributes.

Chapter 11

EXPRESSIONS

Expressions are constructions that evaluate to values. Table 11.1 illustrates a variety of expressions, which, if A = 1 and B = 2, evaluate to the value shown.

Table 11.1. Expressions

Expression	Value
`2`	`2`
`A`	`1`
`A + 2`	`3`
`(A + 2)`	`3`
`(A + 2) * (B - 3)`	`-3`

The operands in an expression may be a value or any other expression. When any operator is applied to an expression, that expression is called an operand. With parentheses for grouping and operators that use other expressions, you can construct expressions as long and complicated as desired.

The available operators, in the order in which they are applied, are as follows:

1. Unary

   `NOT [ADR ADS]`

2. Multiplying

   `* / DIV MOD AND (ISR SHL SHR)`

3. Adding

```
+ - OR (XOR)
```

4. Relational

```
= <> <= >= < > IN
```

Operators shown in parentheses are available only at the extend level of MS-Pascal, those in brackets only at the system level.

A Pascal expression is either a value or the result of applying an operator to one or two values. Although a value can be of almost any type, most MS-Pascal operators only apply to the following types:

```
INTEGER        INTEGER4
WORD           BOOLEAN
REAL           SET
```

The relational operators also apply for the CHAR, enumerated, string, and reference types. For all operators (except the set operator IN), operands must have compatible types.

Simple Type Expressions

As a rule, the operands and the value resulting from an operation are all of the same type. Occasionally, however, the type of an operand is changed to the type required by an operator.

This conversion occurs on two levels: one for constant operands only, and one for all operands. INTEGER to WORD conversion occurs for constant operands only; conversion from INTEGER to REAL and from INTEGER or WORD to INTEGER4 occurs for all operands.

If necessary in constant expressions, INTEGER values change to WORD type. Be careful when mixing INTEGER and WORD constants in expressions. For example, if CBASE is the constant 16#C000 and DELTA is the constant -1, the following expression gives a WORD overflow:

```
WRD (CBASE) + DELTA
```

The overflow occurs because DELTA is converted to the WORD value 16#FFFF, and 16#C000 plus 16#FFFF is greater than MAXWORD. However, the following would work:

```
WRD (ORD (CBASE) + DELTA)
```

This expression gives the INTEGER value -16385, which changes to WORD 16#BFFF. If conversion is needed by an operator or for an assignment, the compiler makes the following conversions:

1. from INTEGER to REAL or INTEGER4
2. from WORD to INTEGER4

The following rules determine the type of the result of an expression involving these simple types:

1. `+ - *`

   These operators operate on INTEGERs, REALs, WORDs, and INTEGER4s, as shown in the following examples:

```
+123
 A + 123
 -23.4
 A - 8
 A * B * 3
```

Mixtures of REALs with INTEGERs and of INTEGER4s with INTEGERs or WORDs are allowed. Where both operands are of the same type, the result type is the type of the operands. If either operand is REAL, the result type is REAL; otherwise, if either operand is INTEGER4, the result type is INTEGER4.

Unary plus (+) and minus (-) are supported, along with the binary forms. Unary minus on a WORD type is 2's complement (NOT is 1's complement); since there are no negative WORD values, this always generates a warning.

Because unary minus has the same precedence level as the adding operators:

```
(X * -1)      {Is illegal}

(-256 AND X) {Is interpreted as -(256 AND X)}
```

2. `/`

   This is a "true" division operator. The result is always REAL. Operands may be INTEGER or REAL (not WORD or INTEGER4).

   Examples of division:

```
34 / 26.4 = 1.28787. . .
18 / 6    = 3.00000. . .
```

3. DIV MOD

These are the operators for integer divide quotient and remainder, respectively. The left operand (dividend) is divided by the right operand (divisor).

Examples of integer division:

```
 123 MOD 5 =  3
-123 MOD 5 = -3 {Sign of result is}
                {sign of dividend }
 123 MOD -5 = 3
 1.3 MOD 5      {Illegal with REAL operands}
 123 DIV 5 = 24
 1.3 DIV 5      {Illegal with REAL operands}
```

Both operands must be of the same type: INTEGER, WORD, or INTEGER4 (not REAL). The sign of the remainder (MOD) is always the sign of the dividend.

MS-Pascal differs from the current draft ISO standard with respect to the semantics for DIV and MOD with negative operands, but the resulting code is more efficient. Programs intended to be portable should not use DIV and MOD unless both operands are positive.

4. AND OR XOR NOT

These extend level operators are bitwise logical functions. Operands must be INTEGER or WORD or INTEGER4 (never a mixture), and cannot be REAL. The result has the type of the operands.

NOT is a bitwise one's complement operation on the single operand. If an INTEGER variable V has the value MAXINT, NOT V gives the illegal INTEGER value -32768. This generates an error if the initialization switch is on and the value is used later in a program.

Given the following initial INTEGER values,

```
X = 2#1111000011110000
Y = 2#1111111100000000
```

AND, OR, XOR, and NOT perform the following functions:

```
X AND Y        1111000011110000
               1111111100000000
               ----------------
               1111000000000000
X OR Y         1111000011110000
               1111111100000000
               ----------------
               1111111111110000
X XOR Y        1111000011110000
               1111111100000000
               ----------------
               0000111111110000
NOT X          1111000011110000
               ----------------
               0000111100001111
```

5. SHL SHR ISR

These extend level operators provide bitwise shifting functions.

SHL and SHR are logical shifts left and right. ISR is an integer (signed) arithmetic shift right: the sign bit is always propagated, even on a WORD type operand. Since the compiler cannot generate a simple right shift for INTEGER division (-1 DIV 2 would be incorrect) and division is a very time-consuming operation, SHR or ISR could be used instead of DIV where appropriate.

Operands must be both INTEGER, both WORD, or both INTEGER4; they cannot be REAL. The result has the same type as the operands.

The left operand is shifted, and the right operand is the shift count in bits. A shift count less than 0 or greater than 32 produces undefined results and generates an error message if the range checking switch is on. Shifts never cause overflow errors; shifted bits are simply lost.

Given that X = 2#1111111100000000, the shifting functions would perform the following operations:

```
X              1111111100000000
X SHL 1        1111111000000000
X SHR 1        0111111110000000
X ISR 1        1111111110000000 {sign extension}
```

Boolean Expressions

The Boolean operators at the standard level of MS-Pascal are:

```
NOT     AND     OR
=       <       >
<>      <=      >=
```

XOR is available at the extend and systems levels.

You may also use P <> Q as an exclusive OR function. Since FALSE < TRUE, P <= Q denotes the Boolean operation "P implies Q." Furthermore, the Boolean operators AND and OR are not the same as the WORD and INTEGER operators of the same name that are bitwise logical functions. The Boolean AND and OR operators may or may not evaluate their operations. The following example illustrates the danger of assuming that they don't:

```
WHILE (I <= MAX) AND (V [I] <> T) DO I := I + 1;
```

If array V has an upper bound MAX, then the evaluation of V [I] for I > MAX is a runtime error. This evaluation may or may not take place. Sometimes both operands are evaluated during optimization, and sometimes the evaluation of one may cause the evaluation of the other to be skipped. In the latter case, either operand may be evaluated first.

Instead, use the following construction:

```
WHILE I <= MAX DO
   IF V [I] <> T THEN I := I + 1 ELSE BREAK;
```

See Chapter 12, "Sequential Control," for information on using AND THEN and OR ELSE to handle situations, such as the previous example, where tests are examined sequentially.

The relational operators produce a Boolean result. The types of the operands of a relational operator (except for IN) must be compatible. If they are not compatible, one must be REAL and the other compatible with INTEGER.

Reference types can only be compared with = and <>. To compare an address type with one of the other relational operators, you must use address field notation, as shown:

```
IF (A.R < B.R) THEN <statement>;
```

Except for the string types STRING and LSTRING, you cannot compare files, arrays, and records as wholes. Two STRING types must have the same upper bound to be compared; two LSTRINGs may have different upper bounds.

In LSTRING comparison, characters past the current length are ignored. If the current length of one LSTRING is less than the length of the other and all characters up to the length of the shorter are equal, the compiler assumes the shorter one is "less than" the longer one. However, two LSTRINGs are not considered equal unless all current characters are equal and their current lengths are equal.

The six relational operators =, <>, <=, >=, <, and > have their normal meaning when applied to numeric, enumerated, CHAR, or string operands. The "Set Expressions" section which follows discusses the meaning of these relational operators (along with the relational operator IN) when applied to sets. Since the relational operators in Boolean expressions have a lower precedence than AND and OR, the following is incorrect:

```
IF I < 10 AND J = K THEN
```

Instead, you must write:

```
IF (I < 10) AND (J = K) THEN
```

Also, you may not use the numeric types where a Boolean operand is called for. (Some other languages permit this.) For an integer I, the clause IF I THEN is illegal; you must use the following instead:

```
IF I <> 0 THEN
```

Note, however, that MS-Pascal does allow the following:

```
$IF I $THEN
```

The inclusion of special "not-a-number" (NaN) values means that a comparison between two real numbers can have a result other than less-than, equal, or greater-than. The numbers can be unordered, meaning one or both are NaNs. An unordered result is the same as "not equal, not less than, and not greater than."

For example, if variables A or B are NaN values:

1. A < B is false.
2. A <= B is false.
3. A > B is false.
4. A >= B is false.
5. A = B is false.
6. A <> B is, however, true.

REAL comparisons do not follow the same rules as other comparisons in many ways. A < B is not always the same as NOT (B <= A); this prevents some optimizations otherwise done by the compiler. If A is a NaN, then A <> A is true; in fact, this is a good way to check for a NaN value.

Set Expressions

Table 11.2 shows the MS-Pascal operators that apply differently to sets than to other types of expressions.

Table 11.2. Set Operators

Operator	Meaning in Set Operations
+	Set union
-	Set difference
*	Set intersection
=	Test set equality
<>	Test set inequality
<= and >=	Test subset and superset
< and >	Test proper subset and superset
IN	Test set membership

Any operand whose type is SET OF S, where S is a subrange of T, is treated as if it were SET OF T. (T is restricted to the range from 0 to 255 or the equivalent ORD values.) Either both operands must be PACKED or neither must be PACKED, unless one operand is a constant or constructed set.

With the IN operator, the left operand (an ordinal) must be compatible with the base type of the right operand (a set). The expression X IN B is TRUE if X is a member of the set B, and FALSE otherwise. X can be outside of the range of the base type of B legally. For example, X IN B is always false if the following statements are true:

```
X = 1
B = SET OF 2..9
```

(1 is compatible, but not assignment compatible, with 2..9).

Angle brackets are set operators only at the extend level of MS-Pascal, since the ISO standard does not support them for sets. They test that a set is a proper subset or superset of another set. Proper subsetting does not permit a set as a subset if the two sets are equal.

Expressions involving sets may use the "set constructor," which gives the elements in a set enclosed in square brackets. Each element can be an expression whose type is in the base type of the set or the lower and upper bounds of a range of elements in the base type. Elements cannot be sets themselves.

Examples of sets involving set constructors:

```
SET_COLOR := [RED, BLUE..PURPLE] - [YELLOW]
SET_NUMBER :=
    [12, J+K, TRUNC (EXP (X))..TRUNC (EXP (X+1))]
```

Set constructor syntax is similar to CASE constant syntax. If X > Y then [X..Y] denotes the empty set. Empty brackets also denote the empty set and are compatible with all sets. Also, if all elements are constant, a set constructor is the same as a set constant.

Like other structured constants, the type identifier for a constant set can be included in a set constant, as in COLORSET [RED..BLUE]. This does not mean that a set constructor with variable elements can be given a type in an expression: NUMBERSET [I..J] is illegal if I or J is a variable.

A set constructor such as [I, J,..K] or an untyped set such as [1, 5..7], is compatible with either a PACKED or an unpacked set. A typed set constant, such as DIGITS [1, 5..7], is only compatible with sets that are PACKED or unpacked, respectively, in the same way as the explicit type of the constant.

Function Designators

A function designator specifies the activation of a function. It consists of the function identifier, followed by a (possibly empty) list of "actual parameters" in parentheses:

```
{Declaration of the function ADD.}
FUNCTION ADD (A, B: INTEGER); INTEGER;
.
.
{Use of the function ADD in an expression.}
X := ADD (7, X * 4) + 123;
{ADD is function designator.}
```

These actual parameters substitute, position for position, for their corresponding "formal parameters," defined in the function declaration.

Parameters can be variables, expressions, procedures, or functions. If the parameter list is empty, the parentheses must be omitted. (See Chapter 13, "Procedure and Function Parameters," for more information on parameters.)

The order of evaluation and binding of the actual parameters varies, depending on the optimizations used. If the $SIMPLE metacommand is on, the order is left to right.

In most computer languages, functions have two different uses:

1. In the mathematical sense, they take one or more values from a domain to produce a resulting value in a range. In this case, if the function never does anything else (such as assign to a global variable or do input/output), it is called a "pure" function.
2. The second type of function may have side effects, such as changing a static variable or a file. Functions of this second kind are said to be "impure."

At the standard level, a function may return either a simple type or a pointer. At the extend level, a function can return any assignable type (i.e., any type except a file or super array).

At the standard level, a pointer returned by a function can only be compared, assigned, or passed as a value parameter. At the extend level, however, the usual selection syntax for reference types, arrays, and records is allowed, following the function designator. See Chapter 10, "Using Variables and Values," for information.

Examples of function designators:

```
SIN (X+Y)
NEXTCHAR
NEXTREC (17)^
{Here the function return type}
{is a pointer, and the returned}
{pointer value is dereferenced.}
NAD.NAME [1]
{Here the function has no parameters.}
{The return type is a record, one}
{field of which is an array.}
{The identifier for that field is}
{NAME. The example above selects}
{the first array component of the}
{returned record.}
```

It is more efficient to return a component of a structure than to return a structure and then use only one component of it. The compiler treats a function that returns a structure like a procedure, with an extra VAR parameter representing the result of the function. The function's caller allocates an unseen variable (on the stack) to receive the return value, but this "variable" is only allocated during execution of the statement that contains the function invocation.

Evaluating Expressions

In cases of ambiguity, an operator at a higher level is applied before one at a lower level. For instance, the following expression evaluates to 7 and not to 9:

```
1 + 2 * 3
```

Use parentheses to change operator precedence. Thus, the following evaluates to 9 rather than 7:

```
(1 + 2) * 3
```

If the $SIMPLE switch is on, sequences of operators of the same precedence are executed from left to right. If the switch is off, the compiler may rearrange expressions and evaluate common subexpressions only once, in order to generate optimized code. The semantics of the precedence relationships are retained, but normal associative and distributive laws are used. For example,

```
X * 3 + 12
```

is an optimization of:

```
3 * (6 + (X - 2))
```

These optimizations may occasionally give you unexpected overflow errors. For example,

```
(I - 100) + (J - 100)
```

will be optimized into the following:

```
(I + J) - 200
```

This may result in an overflow error, although the original expression did not (e.g., if "I" and "J" were each 16400).

An expression in your source file may or may not actually be evaluated when the program runs. For example, the expression F(X + Y)*0 is always zero, so the subexpression (X + Y) and the function call need not be executed.

The compiler does not optimize real expressions as much as, for example, integer expressions, to make sure that the result of a real expression is always what a simple evaluation of the expression, as given, would be. For example, the integer expression

```
((1 + I) - 1) * J
```

is optimized to:

```
I * J
```

but the same expression with real variables is not optimized since the results may be different due to precision loss. Common subexpressions, such as 2 * X in SIN (2 * X) * COS (2 * X), may still be calculated just once and reloaded as necessary, but they are saved in a special 80-bit intermediate precision.

The order of evaluation may be fixed by parentheses:

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

is evaluated by adding A and B first, but

```
A + B + C
```

may be evaluated by adding A and B, B and C, or even A and C first.

Any expression can be passed as a CONST or CONSTS parameter or have its "address" found. The expression is calculated and stored in a temporary variable on the stack, and the address of this temporary variable can be used as a reference parameter or in some other address context.

To avoid ambiguities, enclose such an expression with operators or function calls in parentheses. For example, to invoke a procedure FOO (CONST X, Y: INTEGER), FOO (I, (J+14)) must be used instead of FOO(I, J+14).

This implies a subtle distinction in the case of functions. For example:

```
FUNCTION SUM (CONST A, B: INTEGER): INTEGER;
BEGIN
SUM := A;
IF B <> 0 THEN
  SUM := SUM (SUM, (SUM (B, 0) - 1)) + 1;
END;
```

In this example, SUM is called recursively subtracting one from B until B is zero.

The use of a function identifier in a WITH statement follows a similar rule. For example, given a parameterless function, COMPLEX, which returns a record, "WITH COMPLEX" means "WITH the current value of the function." This can only occur inside the COMPLEX function itself. However, "WITH (COMPLEX)" causes the function to be called and the result assigned to a temporary local variable.

Another way to describe this is to distinguish between "address" and "value" phrases. The left-hand side of an assignment, a reference parameter, the ADR and ADS operators, and the WITH statement all need an address. The right-hand side of an assignment and a value parameter all need a value.

If an address is needed but only a value, such as a constant or an expression in parentheses, is available, the value must be put into memory so it has an address. For constants, the value goes in static memory; for expressions, the value goes in stack (local) memory. A function identifier refers to the current value of the function as an address, but causes the function to be called as a value.

Finally, in the scope of a function, the intrinsic procedure RESULT permits a reference to the current value of a function instead of invoking it recursively. For a function F, this means ADR F and ADR RESULT (F) are the same: the address of the current value of F.RESULT forces use of the current value in the same way that putting the function in parentheses, as in (F(X)), forces evaluation of the function.

Other Features of Expressions

EVAL and RESULT are two procedures available at the extend level for use with expressions. EVAL obtains the effect of a procedure from a function; RESULT yields the current value of a function within a function or nested procedure or function.

At the system level, the function RETYPE allows you to change the type of a value.

The EVAL Procedure

EVAL evaluates its parameters without actually calling anything. Generally, you use EVAL to obtain the effect of a procedure from a function. In such cases, the values returned by functions are of no interest, so EVAL is only useful for functions with side effects. For example, a function that advances to the next item and also returns the item might be called in EVAL just to advance to the next item, since there is no need to obtain a function return value.

Examples of the EVAL procedure:

```
EVAL (NEXTLABEL (TRUE))
EVAL (SIDEFUNC (X, Y), INDEX (4), COUNT)
```

The RESULT Function

Within the scope of a function, the intrinsic procedure RESULT permits a reference to the current value of a function instead of invoking it recursively. For a function F, this means ADR F and ADR RESULT (F) are the same; that is, the address of the current value of F. RESULT forces use of the current value in the same way that putting the function in parentheses as in (F (X)) forces evaluation of the function.

Examples of the RESULT function:

```
FUNCTION FACTORIAL (I: INTEGER): INTEGER;
 BEGIN
   FACTORIAL := 1; WHILE I > 1 DO
   BEGIN
     FACTORIAL := I * RESULT (FACTORIAL);
     I := I - 1;
   END;
 END;
FUNCTION ABSVAL (I: INTEGER): INTEGER;
BEGIN
  ABSVAL := I;
  IF I < 0 THEN ABSVAL := -RESULT (ABSVAL);
END;
```

The RETYPE Function

Occasionally, you need to change the type of a value. You can do this with the RETYPE function, available at the system level of MS-Pascal. If the new type is a structure, RETYPE can be followed by the usual selection syntax. You must use RETYPE with caution: it works on the memory byte level and ignores whether the low order byte of a two-byte number comes first or second in memory.

Examples of the RETYPE function:

```
RETYPE (COLOR, 3)               {inverse of ORD}
RETYPE (STRING2, I*J+K) [2]     {effect may vary}
```

Chapter 12

STATEMENTS

The body of a program, procedure, or function contains statements. Statements denote actions that the program can execute. This chapter first discusses the syntax of statements and then separates and describes two categories of statements: simple statements and structured statements. A simple statement has no parts that are themselves other statements; a structured statement consists of two or more other statements. Table 12.1 lists the statements in each category in MS-Pascal.

Table 12.1. Microsoft Pascal Statements

Simple	Structured
Assignment (:=)	Compound
Procedure	IF/THEN/ELSE
GOTO	CASE
BREAK	FOR
CYCLE	WHILE
RETURN	REPEAT
Empty	WITH

The Syntax of Pascal Statements

Pascal statements are separated by a semicolon (;) and enclosed by reserved words such as BEGIN and END. A statement begins, optionally, with a label. Each of these three elements of statement syntax are discussed in the following sections.

Labels

Any statement referred to by a GOTO statement must have a label. A label at the standard level is one or more digits; leading zeros are ignored. Constant identifiers, expressions, and nondecimal notation cannot serve as labels.

All labels must be declared in a LABEL section. At the extend level, a label can also be an identifier.

Example using labels and GOTO statements:

```
PROGRAM LOOPS(INPUT,OUTPUT);
LABEL 1, HAWAII, MAINLAND;

BEGIN
  MAINLAND: GOTO 1;
  HAWAII: WRITELN ('Here I am in Hawaii');
  1: GOTO HAWAII
END.
```

A loop label is any label immediately preceding a looping statement: WHILE, REPEAT, or FOR. At the extend level, a BREAK or CYCLE statement can also refer to a loop label.

Both a CASE constant list and a GOTO label may precede a statement, in which case the CASE constants come first and then the GOTO label. In the following example, 321 is a CASE value, 123 is a label:

```
321: 123: IF LOOP THEN GOTO 123
```

Separating Statements

Semicolons separate statements. Semicolons do not terminate statements. However, since Pascal permits the empty statement, using the semicolon as if it were a statement terminator is rarely disastrous.

Example showing semicolon to separate statements:

```
BEGIN
   10: WRITELN;
   A := 2 + 3;
   GOTO 10
END
```

A common error is to terminate the THEN clause in an IF/THEN/ELSE statement with a semicolon. Thus, the following example generates a warning message:

```
IF A = 2 THEN WRITELN;
ELSE A = 3
```

Another common error is to put a semicolon after the DO in a WHILE or FOR statement:

```
FOR I := 1 TO 10 DO;
BEGIN
  A[I] := I;
  B[I] := 10 - I;
END;
```

The previous example, as written, will "execute" an empty statement ten times, then execute the array assignments once. Since there are occasional legitimate uses for repeating an empty statement, no warning is given when this occurs.

The semicolon also follows the reserved word END at the close of a block of program statements.

The Reserved Words BEGIN and END

Whenever you want a program to execute a group of statements, instead of a single simple statement, you may enclose the block with the reserved words BEGIN and END.

For example, the following group of statements between BEGIN and END will all be executed if the condition in the IF statement is TRUE:

```
IF (MAX > 10) THEN
BEGIN
  MAX = 10;
  MIN = 0;
  WRITELN (MAX,MIN)
END;
WRITELN ('done')
```

At the extend level, you may substitute a pair of square brackets for the pair of keywords BEGIN and END.

Simple Statements

A simple statement is one in which no part constitutes another statement. Simple statements in standard Pascal are:

1. the assignment statement
2. the procedure statement
3. the GOTO statement
4. the empty statement

The empty statement contains no symbols and denotes no action. It is included in the definition of the language primarily to permit you to use a semicolon after the last in a group of statements enclosed between BEGIN and END.

The extend level in MS-Pascal adds three simple statements: BREAK, CYCLE, and RETURN.

Assignment Statements

The assignment statement replaces the current value of a variable with a new value, which you specify as an expression. Assignment is denoted by an adjacent colon and equal sign characters (:=).

Examples of assignment statements:

```
A := B

A[I] := 12 * 4 + (B * C)

X : = Y
{Illegal. Colon (:) and equal}
{sign (=) must be adjacent.}

A + 2 := B
{Illegal. A + 2 is not a variable.}

A := ADD (1,1)
```

The value of the expression must be assignment compatible with the type of the variable. Selection of the variable may involve indexing an array or dereferencing a pointer or address. If it does, the compiler may, depending on the optimizations performed, mix these actions with the evaluation of the expression. If the $SIMPLE metacommand is on, the expression is evaluated first.

An assignment to a nonlocal variable (including a function return) puts an equal sign (=) or percent sign (%) in the G column of the listing file. (See Chapter 17, "Listing File Format," for more information about these and other symbols used in the listing.)

Within the block of a function, an assignment to the identifier of the function sets the value returned by the function. The assignment to a function identifier may occur either within the actual body of the function or in the body of a procedure or function nested within it.

If the range checking switch is on, an assignment to a set, subrange, or LSTRING variable may imply a runtime call to the error checking code.

According to the MS-Pascal optimizer, each section of code without a label or other point that could receive control is eligible for rearrangement and common subexpression elimination. Naturally, the order of execution is retained when necessary.

Given these statements,

```
X := A + C + B;
Y := A + B;
Z := A
```

the compiler might generate code to perform the following operations:

1. Get the value of A and save it.

2. Add the value of B and save the result.

3. Add the value of C and assign it to X.

4. Assign the saved A + B value to Y.

5. Assign the saved A value to Z.

This optimization occurs only if assignment to X and Y and getting the value of A, B, or C are all independent. If C is a function without the PURE attribute and A is a global variable, evaluating C might change A. Then, since the order of evaluation within an expression in this case is not fixed, the value of A in the first assignment could be the old value or the new one.

However, since the order of evaluation among statements is fixed, the value of A in the second and third assignments is the new value.

The following actions may limit the ability of the optimizer to find common subexpressions:

1. assignment to a nonlocal variable
2. assignment to a reference parameter
3. assignment to the referent of a pointer
4. assignment to the referent of an address variable
5. calling a procedure
6. calling a function without the PURE attribute

The optimizer does allow for "aliases," that is, a single variable with two identifiers, perhaps one as a global variable and one as a reference parameter.

Procedure Statements

A procedure statement executes the procedure denoted by the procedure identifier.

For example, assume you have defined the procedure DO_IT:

```
PROCEDURE DO_IT;
BEGIN
  WRITELN('Did it')
END;
```

DO_IT is now a statement that can be executed simply by invoking its name:

```
DO_IT
```

If you declare the procedure with a formal parameter list, the procedure statement must include the actual parameters.

MS-Pascal includes a large number of predeclared procedures. See Chapter 14, "Available Procedures and Functions," for complete information. One of the predeclared procedures is ASSIGN. You need not declare it in order to use it.

```
ASSIGN (INFILE, 'MYFILE')
```

Note that the ASSIGN procedure contains a parameter list. These parameters are the actual parameters that are bound to the formal parameters in the procedure declaration. For a discussion of formal and reference parameters, see Chapter 13, "Procedure and Function Parameters."

The GOTO Statement

A GOTO statement indicates that further processing continues at another part of the program text, namely at the place of the label. You must declare a LABEL in a LABEL declaration section, before using it in a GOTO statement.

Several restrictions apply to the use of GOTO statements:

1. A GOTO must not jump to a more deeply nested statement, that is, into an IF, CASE, WHILE, REPEAT, FOR, or WITH statement. GOTOs from one branch of an IF or CASE statement to another are permitted.

2. A GOTO from one procedure or function to a label in the main program or in a higher level procedure or function is permitted. A GOTO may jump out of one of these statements, so long as the statement is directly within the body of the procedure or function. However, such a jump generates extra code both at the location of the GOTO and at the location of the label. The GOTO and label must be in the same compiland, since labels, unlike variables, cannot be given the PUBLIC attribute.

Examples of GOTO statements, both legal and illegal:

```
PROGRAM LABEL_EXAMPLES;
LABEL 1, 2, 3, 4;

  PROCEDURE ONE;
  LABEL 11, 12, 13;

    PROCEDURE IN_ONE;
    LABEL 21;
    {Outer level GOTOs cannot jump in to 21.}

    BEGIN
      IF TUESDAY THEN GOTO 1
      ELSE GOTO 11;
      {1 and 11 are both legal outer level labels.}
      21: WRITE ('IN_ONE')
    END;

  BEGIN   {Procedure one}
    IF RAINING THEN GOTO 1 ELSE GOTO 11;
    {That was legal.}
    11: GOTO 21;
    {Illegal. Cannot jump into inner level}
    {procedures.}
  END;
```

```
   PROCEDURE TWO;
   BEGIN
    GOTO 11
     {Illegal. Cannot jump into different procedure}
     {at same level.}
   END;

  BEGIN  {Main level}
   IF SEATTLE
   THEN
     BEGIN BEGIN
       GOTO 2;
       {OK to go to 2 at program level.}

       4: WRITE ('here');
     END END
   ELSE GOTO 4;
   {OK to jump into THEN clause.}
   2: GOTO 3;
   {Illegal. Cannot jump into REPEAT statement.}
   REPEAT
       WHILE MS_BYRON DO
         3: GOTO 2
         {OK to jump out of loops.}
   UNTIL DATE;
   1: GOTO 11;
   {Illegal. Cannot jump into procedure from program.`

 END.
```

If the $GOTO metacommand is on, every GOTO statement is flagged with a warning that reminds you that "GOTOs are considered harmful." This may be useful either in an educational environment or for finding all GOTOs in a program in order to locate a bug. The J (jumps) column of the listing file contains the following:

1. A plus (+) or an asterisk (*) flags a GOTO to a label later in the listing.

2. A minus sign (-) or an asterisk (*) marks a GOTO to a label already encountered in the listing.

See Chapter 17, "Listing File Format," for details about the listing file.

The BREAK, CYCLE, and RETURN Statements

At the extend level, BREAK, CYCLE, and RETURN statements are allowed in addition to the simple statements already described. These statements perform the following functions:

1. BREAK exits the currently executing loop.
2. CYCLE exits the current iteration of a loop and starts the next iteration.
3. RETURN exits the current procedure, function, program, or implementation.

All three statements are functionally equivalent to a GOTO statement.

1. A BREAK statement is a GOTO to the first statement after a repetitive statement.
2. A CYCLE statement is a GOTO to an implied empty statement after the body of a repetitive statement. This jump starts the next iteration of a loop. In either a WHILE or REPEAT statement, CYCLE performs the Boolean test in the WHILE or UNTIL clause before executing the statement again; in a FOR statement, CYCLE goes to the next value of the control variable.
3. A RETURN statement is a GOTO to an implied empty statement after the last statement in the current procedure or function or the body of a program or implementation.

The J (jump) column in the listing file contains a plus sign (+) or an asterisk (*) for a BREAK statement, a minus sign (-) or an asterisk (*) for a CYCLE statement, and an asterisk (*) for a RETURN statement.

(See Chapter 17, "Listing File Format," for information about the listing file.)

BREAK and CYCLE have two forms, one with a loop label and one without. If you give a loop label, the label identifies the loop to exit or restart. If you don't give a label, the innermost loop is assumed, as shown in the following example:

```
OUTER: FOR I := 1 TO N1 DO
  INNER: FOR J := 1 TO N2 DO
    IF A [I, J] = TARGET THEN BREAK OUTER;
```

Structured Statements

Structured statements are themselves composed of other statements. There are four kinds of structured statements:

1. compound statements
2. conditional statements
3. repetitive statements
4. WITH statement

The control level is shown in the C (control) column of the listing file. The value in the C column is incremented each time control passes to a nested statement; conversely, this value is decremented each time control passes back to the nesting statement. This helps you search for a missing or extra END in a program.

Compound Statements

The compound statement is a sequence of simple statements, enclosed by the reserved words BEGIN and END. The components of a compound statement execute in the same sequence as they appear in the source file.

Examples of compound statements:

```
BEGIN
   TEMP := A [I];
   A[I] := A [J];
   A [J] := TEMP
   {Semicolon not needed here.}
END

BEGIN
   OPEN_DOOR;
   LET_EM_IN;
   CLOSE_DOOR;
   {Semicolon signifies empty statement.}
END
```

All MS-Pascal conditional and repetitive control structures (except REPEAT) operate on a single statement, not on multiple statements with ending delimiters. In this context, BEGIN and END serve as punctuation, like semicolon, colon, or parentheses. If you prefer, you may substitute a pair of square brackets for the BEGIN and END pair of reserved words. Note that a right bracket (]) matches only a left bracket ([) (not a BEGIN, CASE, or RECORD). In other words, right bracket is not a synonym for END.

Brackets may not be used as synonyms for BEGIN and END to enclose the body of a program, implementation, procedure, or function; only BEGIN and END can be used for this purpose.

Examples of brackets replacing BEGIN and END:

```
IF FLAG THEN [X := 1; Y := -1]
ELSE [X := -1; Y := 0];

WHILE P.N <> NIL DO
   [Q := P; P := P.N; DISPOSE (Q)];

FUNCTION R2 (R: REAL): REAL;
   [R2 := R * 2]
   {Illegal.}
```

Conditional Statements

A conditional statement selects for execution only one of its component statements. The conditional statements are the IF and CASE statements. Use the IF statement for one or two conditions, the CASE statement for multiple conditions.

The IF Statement

The IF statement allows for conditional execution of a statement. If the Boolean expression following the IF is true, the statement following the THEN is executed. If the Boolean expression following the IF is false, the statement following the ELSE, if present, is executed.

Examples of IF statements:

```
IF I > 0 THEN I := I - 1
{No semicolon here.}
ELSE I := I + 1

IF (I <= TOP) AND (ARRI [I] <> TARGET) THEN
   I := I + 1

IF I <= TOP THEN
   IF ARRI [I] <> TARGET THEN
      I := I + 1

IF I = 1 THEN
   IF J = 1 THEN
      WRITELN('I equals J')
   ELSE
      WRITELN('DONE only if I = 1 and J <> 1')
       {This ELSE is paired with the most deeply}
       {nested IF. Thus, the second WRITELN is}
       {executed only if I = 1 and J <> 1.}

IF I = 1 THEN BEGIN
   IF J = 1 THEN WRITELN('I equals J')
   END
ELSE
   WRITELN('DONE only if I <> 1')
       {Now the ELSE is paired with the first IF,}
       {since the second IF statement is}
       {bracketed by the BEGIN/END pair. Thus,}
       {the second WRITELN is executed if I <> 1.}
```

A semicolon (;) preceding an ELSE is always incorrect. The compiler skips it during compilation and issues a warning message.

The Boolean expression following an IF may include the sequential control operators described in Chapter 12, "Sequential Control," later in this chapter.

The CASE Statement

The CASE statement consists of an expression (called the CASE index) and a list of statements. Each statement is preceded by a constant list, called a CASE constant list. The one statement executed is the one whose CASE constant list contains the current value of the CASE index. The CASE index and all constants must be of compatible, ordinal types.

Examples of CASE statements:

```
CASE OPERATOR OF
  PLUS: X := X + Y;
  MINUS: X := X - Y;
  TIMES: X := X * Y
END
{OPERATOR is the CASE index. PLUS, MINUS, and}
{TIMES are CASE constants. In this instance,}
{they are all of the values assumable by the}
{enumerated variable, OPERATOR.}

CASE NEXTCH OF
  'A'..'Z', '_' : IDENTIFIER;
  '+', '-', '*', '/' : OPERATOR;
  {Commas separate CASE constants}
  {and ranges of CASE constants.}
  OTHERWISE
    WRITE ('Unknown Character')
    {i.e., if any other character}
END
```

The CASE constant syntax is the same as for RECORD variant declarations. In standard Pascal, a CASE constant is one or more constants separated by commas. At the extend level, you may substitute a range of constants, such as 'A'..'Z', for a constant. No constant value can apply to more than one statement. The extend level also allows the CASE statement to end with an OTHERWISE clause. The OTHERWISE clause contains additional statements to be executed in the event that the CASE index value is not in the given set of CASE constant values. One of two things happens if the CASE index value is not in the set and no OTHERWISE clause is present:

1. If the range checking switch is on, a runtime error is generated.
2. If the range checking switch is off, the result is undefined (and may be catastrophic).

In MS-Pascal, control does not automatically pass to the next executable statement as in UCSD Pascal and some other languages. If you want this effect, include an empty OTHERWISE clause.

A semicolon (;) may appear after the final statement in the list, but is not required. The compiler skips over a colon (:) after an OTHERWISE and issues a warning.

Depending on optimization, the code generated by the compiler for a CASE statement may be either a "jump table" or series of comparisons (or both). If it is a jump table, a jump to an arbitrary location in memory can occur if the control variable is out of range and the range checking switch is off.

Repetition Statements

Repetition statements specify repeated execution of a statement. In standard Pascal, these include the WHILE, REPEAT, and FOR statements.

At the extend level in MS-Pascal, there are two additional statements, BREAK and CYCLE, for leaving or restarting the statements being repeated. These statements are functionally equivalent to a GOTO but easier to use.

The WHILE Statement

The WHILE statement repeats a statement zero or more times, until a Boolean expression becomes false.

Examples of WHILE statements:

```
WHILE P <> NIL DO P := NEXT (P)

WHILE NOT MICKEY DO
   BEGIN
      NEXTMOUSE;
      MICE := MICE + 1
   END
```

The Boolean expression in a WHILE statement may include the sequential control operators described in Chapter 12, "Sequential Control."

Use WHILE if it is possible that no iterations of the loop may be necessary; use REPEAT where you expect that at least one iteration of the loop is required.

The REPEAT Statement

The REPEAT statement repeats a sequence of statements one or more times, until a Boolean expression becomes true.

Examples of REPEAT statements:

```
REPEAT
   READ (LINEBUFF);
   COUNT := COUNT + 1
UNTIL EOF;

REPEAT GAME UNTIL TIRED;
```

The Boolean expression in a REPEAT statement may include the sequential control operators described in Chapter 12, "Sequential Control."

Use the REPEAT statement to execute statements, not just a single statement, one or more times until a condition becomes true. This differs from the WHILE statement in which a single statement may not be executed at all.

The FOR Statement

The FOR statement tells the compiler to execute a statement repeatedly while a progression of values is assigned to a variable, called the control variable of the FOR statement. The values assigned start with a value called the initial value and end with one called the final value.

The FOR statement has two forms, one where the control variable increases in value and one where the control variable decreases in value:

```
FOR I := 1 TO 10 DO
{I is the control variable.}
  SUM := SUM + VICTORVECTOR [I]

FOR CH := 'Z' DOWNTO 'A' DO
{CH is the control variable.}
  WRITE (CH)
```

You may also use a FOR statement to step through the values of a set, as shown:

```
FOR TINT :=
   LOWER (SHADES) TO UPPER (SHADES) DO
      IF TINT IN SHADES
      THEN PAINT_AREA (TINT);
```

The ISO standard gives explicit rules regarding the control variable in FOR statements:

1. It must be of an ordinal type.
2. It must also be an entire variable, not a component of a structure.
3. It must be local to the immediately enclosing program, procedure, or function and cannot be a reference parameter of the procedure or function.

   However, at the extend level of MS-Pascal, the control variable may also be any STATIC variable, such as a variable declared at the program level, unless the variable has a segmented ORIGIN attribute. Using a program level variable is an ISO error not caught.

4. No assignments to the control variable are allowed in the repeated statement. This error is caught by making the control variable READONLY within the FOR statement; it is not caught when a procedure or function invoked by the repeated statement alters the control variable. The control variable cannot be passed as a VAR (or VARS) parameter to a procedure or function.
5. The initial and final values of the control variable must be compatible with the type of the control variable. If the statement is executed, both the initial and final values must also be assignment compatible with the control variable. The initial value is always evaluated first, and then the final value. Both are evaluated only once before the statement executes.

The statement following the DO is not executed at all if:

1. The initial value is greater than the final value in the TO case.
2. The initial value is less than the final value in the DOWNTO case.

The sequence of values given the control variable starts with the initial value. This sequence is defined with the SUCC function for the TO case or the PRED function for the DOWNTO case until the last execution of the statement, when the control variable has its final value. The value of the control variable, after a FOR statement terminates naturally (whether or not the body executes), is undefined. It may vary due to optimization and, if $INITCK is on, may be set to an uninitialized value. However, the value of the control variable after leaving a FOR statement with GOTO or BREAK is defined as the value it had at the time of exit.

In standard Pascal, the body of a FOR statement may or may not be executed, so a test is necessary to see whether the body should be executed at all. However, if the control variable is of type WORD (or a subrange) and its initial value is a constant zero, the body must be executed no matter what the final value. In this case, no extra test need be executed and no code is generated to perform such a test.

Also, a control variable with the STATIC attribute may be more efficient than one that is not.

At the extend level in MS-Pascal, you may use temporary control variables:

```
FOR VAR <control-variable>
```

The prefix VAR causes the control variable to be declared local to the FOR statement (i.e., at a lower scope) and need not be declared in a VAR section. Such a control variable is not available outside the FOR statement, and any other variable with the same identifier is not available within the FOR statement itself. Other synonymous variables are, however, available to procedures or functions called within the FOR statement.

Examples of temporary control variables:

```
FOR VAR I := 1 TO 100 DO
    SUM := SUM + VICTOR [I]

FOR VAR COUNTDOWN := 10 DOWNTO LIFT_OFF DO
    MONITOR_ROCKET
```

The BREAK and CYCLE Statements

In theory, a program using the MS-Pascal extend level BREAK and CYCLE statements does not need to use any GOTO statements.

Each of these two statements has two forms, one with a loop label and one without. A loop label is a normal GOTO label prefixed to a FOR, WHILE, or REPEAT statement. Since, at the extend level, you may use identifier labels, a suggested practice is to use integers for labels referenced by GOTOs and identifiers for loop labels.

Examples of CYCLE and BREAK statements:

```
LABEL SEARCH, CLIMB;
  .
  .
SEARCH: WHILE I <= I_TOP DO
  IF PILE [I] = TARGET THEN BREAK SEARCH
  ELSE I := I + 1;
  .
  .
FOR I := 1 TO N DO
  IF NEXT [I] = NIL THEN BREAK;
  .
  .
CLIMB: WHILE NOT ITEM^.LEAF DO
  BEGIN
    IF ITEM^.LEFT <> NIL
      THEN [ITEM := ITEM^.LEFT; CYCLE CLIMB];
    IF ITEM^.RIGHT <> NIL
      THEN [ITEM := ITEM^.RIGHT; CYCLE CLIMB];
    WRITELN ('Very strange node');
    BREAK CLIMB
  END;
```

The WITH Statement

The WITH statement opens the scope of a statement to include the fields of one or more records, so you can refer to the fields directly. For example, the following statements are equivalent:

```
WITH PERSON DO WRITE (NAME, ADDRESS, PHONE)
WRITE (PERSON.NAME, PERSON.ADDRESS, PERSON.PHONE)
```

The record given may be a variable, constant identifier, structured constant, or function identifier; it may not be a component of a PACKED structure. If you use a function identifier, it refers to the function's local result variable. If the record given in a WITH statement is a file buffer variable, the compiler issues a warning, since changing the position in the WITH statement may cause an error.

The record given can also be any expression in parentheses, in which case the expression is evaluated and the result assigned to a temporary (hidden) variable. If you want to evaluate a function designator, you must enclose it in parentheses.

You may give a list of records after the WITH, separated by commas. Each record so listed must be of a different type from all the others, since the field identifiers refer only to the last instance of the record with the type. These statements are equivalent:

```
WITH PMODE, QMODE DO statement
WITH PMODE DO WITH QMODE DO statement
```

Any record variable of a WITH statement that is a component of another variable is selected before the statement is executed. Active WITH variables should not be passed as VAR or VARS parameters, nor can their pointers be passed to the DISPOSE procedure. However, these errors are not caught by the compiler. Assignments to any of the record variables in the WITH list or components of these variables are allowed, as long as the WITH record is a variable.

In MS-Pascal, every WITH statement allocates an address variable that holds the address of the record. If the record variable is on the heap, the pointer to it should not be DISPOSEd within the WITH statement. If the record variable is a file buffer, no I/O should be done to the file within the WITH statement. Avoid assignments to the WITH record itself in programs intended to be portable.

Sequential Control

To increase execution speed or guarantee correct evaluation, it is often useful in IF, WHILE, and REPEAT statements to treat the Boolean expression as a series of tests. If one test fails, the remaining tests are not executed. Two extend level operators in MS-Pascal provide for such tests:

1. AND THEN

   X AND THEN Y is false if X is false; Y is only evaluated if X is true.

2. OR ELSE

   X OR ELSE Y is true if X is true; Y is only evaluated if X is false.

If you use several sequential control operators, the compiler evaluates them strictly from left to right.

You may only include these operators in the Boolean expression of an IF, WHILE, or UNTIL clause; they may not be used in other Boolean expressions. Furthermore, they may not occur in parentheses and are evaluated after all other operators.

Examples of sequential control operators:

```
IF SYM <> NIL AND THEN SYM^.VAL < 0 THEN
   NEXT_SYMBOL

WHILE I <= MAX AND THEN VECT [I] <> KEY DO
   I := I + 1;

REPEAT GEN (VAL)
UNTIL VAL = 0 OR ELSE (QU DIV VAL) = 0;

WHILE POOR AND THEN GETTING_POORER
   OR ELSE BROKE AND THEN BANKRUPT DO
      GET_RICH
```

INDEX

This is a complete index for the *Microsoft Pascal Reference Manual,* Parts 1 and 2. Pages i through xix and 1 through 174 reference Part 1, which contains the introduction and Chapters 1 through 12. Pages 187 through 440 reference Part 2, which contains Chapters 13 through 17 and Appendices A through H.

! comment to end-of-line, 2-3
radix separator, 9-6
* operator, 9-11, 11-3, 11-8
+ operator, 11-3, 11-8
− operator, 11-3, 11-8
. period, 1-3
/ operator, 11-3
:= assignment, 12-4
< operator, 11-7, 11-8
<= operator, 11-7, 11-8
= operator, 11-7, 11-8
< > operator, 11-7, 11-8
> operator, 11-7, 11-8
>= operator, 11-7, 11-8
? question mark, 2-6
[ ] square brackets, 11-8
_ underscore, Intro 3

A

A2DRQQ function, ... 14-11
A2SRQQ function, ... 14-11
ABORT procedure, ... 14-9, 17-5
ABS function, ... 14-9
ACDRQQ function, ... 14-9
ACSRQQ function, ... 14-9
Actual parameters, ... 11-9
Address types, ... Intro 3
ADR operator, ... 11-1
ADRMEM predeclared type, ... 8-5
ADS operator, ... 11-1
ADSMEM predeclared type, ... 8-5
AIDRQQ function, ... 14-10
AISRQQ function, ... 14-10
Aliases, ... 12-6
ALLHQQ function, ... 14-10
AND operator, ... 11-4
AND THEN operator, ... 12-19
ANDRQQ function, ... 14-10
ANSRQQ function, ... 14-10
ARCTAN function, ... 14-10
Arithmetic functions, ... 14-4
Arrays,
- component type, ... 6-2
- conformant, ... 6-4
- dynamic, ... 6-4
- index type, ... 6-2
- n-dimensional, ... 6-2
- super, ... 6-3

Arrows in syntax diagrams, ... A-1
ASCII
- character codes, ... D-1
- character set, ... 2-1, 5-3
- files, ... 7-4

ASDRQQ function, ... 14-10
ASSIGN procedure, ... 14-10, 15-23, 16-3
Assignment
- compatibility, ... 4-6
- statement, ... 12-4

ASSRQQ function, ... 14-10
ATDRQQ function, ... 14-11
ATSRQQ function, ... 14-11

Attributes
EXTERN, ... 10-10
EXTERNAL, ... 10-10
FORTRAN, ... 13-10
functions, ... 13-5
INTERRUPT, ... 13-11
names, ... E-2
ORIGIN, ... 10-11, 13-10
PORT, ... 10-11
procedures, ... 13-5
PUBLIC, ... 10-10, 13-9
PURE, ... 13-12
READONLY, ... 10-12
STATIC, ... 10-9
variables, ... 10-8
Available procedures, functions, ... F-1

B

Back end compiler errors, ... H-31
Base type, ... 6-18
BEGIN, ... 12-3
BEGOQQ procedure, ... 14-11
BEGXQQ procedure, ... 14-11
Bibliography of tutorial references, ... Intro 5
BINARY files, ... 7-4
Boolean
expressions, ... 11-6
operator, ... 11-6
type, ... 5-3
Boxes in syntax diagrams, ... A-1
BRAVE metacommand, ... 17-5, 17-17
BREAK statement, ... Intro 4, 12-9, 12-17
Buffer variable, ... 7-3, 10-2
BYLONG function, ... 14-12
BYTE predeclared subrange, ... Intro 3, 5-5
BYWORD function, ... 14-12

C

CASE
constant, ... 6-15, 12-13
index, ... 12-13
statement, ... 12-13
CHAR type, ... 5-3
Character strings, ... 9-7
CHDRQQ function, ... 14-12
CHR function, ... 14-12
CHSRQQ function, ... 14-12

Circles in syntax diagram, ... A-1
CLOSE procedure, ... 14-12, 15-23
CNDRQQ function, ... 14-12
CNSRQQ function, ... 14-12
Codes, error, ... F-1
Comments, ... 2-2
Compatibility
- assignment, ... 4-6
- of types, ... 4-5

Compilable parts of programs, ... 1-2
Compilands, ... 1-2
Compile time errors, ... F-1
Compiler errors
- errors, ... H-31, H-32
- switches, ... 17-18

Compiling programs, ... 16-1
Component
- type, ... 6-2
- variable, ... 10-3

Compound statements, ... 12-10
CONCAT procedure, ... 14-13
Concurrent I/O, ... 7-4, 15-9
Conditional
- compilation, ... Intro 4
- statement, ... 12-11

Conformant array, ... 6-4
Constants
- declarations, ... 9-2
- enumerated, ... 9-2
- expressions, ... 1-10, 9-7, 9-10
- identifiers, ... 1-10, 9-2
- length restriction, ... 9-2
- parameters, ... 13-17
- string, ... 1-10, 9-7
- structured, ... Intro 3, 1-10, 9-8

CONST section, ... 1-10
CONSTS parameters, ... 8-6, 13-17
Control
- sequential, ... 12-19

COPYLST procedure, ... 14-13
COPYSTR procedure, ... 14-13
COS function, ... 14-13
CYCLE statement, ... Intro 4, 12-9, 12-17

D

DATE procedure, ... 14-13
DEBUG metacommand, ... 17-5
Debugging metacommands, ... 17-4
Declarations
- flexible order of, ... 9-3, 13-2
- FORWARD, ... 13-7
- function, ... 13-1, 13-3
- identifiers, ... 3-3
- procedure, ... 13-1
- type, ... 4-3
- variables, ... 10-2

DECODE function, ... 14-13
DELETE procedure, ... 14-14
Diagrams of MS-Pascal syntax, ... A-1
Digits, ... 2-2
DIRECT files, ... Intro 3, 7-7
Directives, ... E-2
- EXTERN, ... 13-5, 13-8
- EXTERNAL, ... 13-8
- function, ... 13-5
- FORWARD, ... 13-5, 13-7
- procedure, ... 13-5

DISBIN procedure, ... 13-12, 14-14
DISCARD procedure, ... 14-14, 15-24
DISPOSE procedure, ... 14-14
DIV operator, ... 11-4
DO in structured constants, ... 9-9
Dynamic array, ... 6-4

E

Empty
- statement, ... 12-4
- variant, ... 6-15

EMSEQQ routine, ... 17-5
ENABIN procedure, ... 13-12, 14-15
ENCODE function, ... 14-15
END keyword, ... 12-3
ENDOQQ procedure, ... 14-15
ENDXQQ procedure, ... 14-16

ENTGQQ identifier, 16-3
Entire variable, 10-4
ENTRY metacommand, 17-5
Enumerated
    constants, 9-1
    types, 5-3
EOF function, 14-16, 15-2, 15-6
EOLN function, 14-16, 15-2, 15-6
Errors
    conditions, H-1
    file system, H-32, H-35
    front end, H-2
    INTEGER4, H-43
    memory, H-37
    miscellaneous, H-43
    operating system, H-34
    ordinal arithmetic, H-38
    panic, H-2
    REAL arithmetic, H-40
    runtime file system, H-32, H-37
    structured types, H-42
ERRORS metacommand, 17-6, 17-17
EVAL procedure, 11-13, 14-16
EXDRQQ function, 14-16
EXP function, 14-17
Explicit field offsets, 6-16
Expressions
    basic types, 1-8
    compatibility, 4-5
    constant, 9-10
    simple, 11-2
    values, 11-1
EXSRQQ function, 14-16
Extend level
    CASE constant, 12-13
    function, 10-4
    intrinsics, 14-6
    I/O, 7-8, 15-23
    Pascal, Intro 2
EXTEND metacommand, 17-3
EXTERN attribute, 10-10
EXTERN directive, Intro 3, 13-5, 13-8
EXTERNAL attribute, 10-10
EXTERNAL directive, 13-8

F

FCBFQQ, ... 7-8
Features of MS-Pascal, ... Intro 3, B-5
Fields
  identifiers, ... 10-5
  record components, ... 6-13
  variables, ... 10-5
File system errors, ... H-32, H-35
Files
  buffer variable, ... 10-2, 10-5
  FCB fields, ... 7-8
  FCBFQQ, ... 7-8
  INPUT, ... 7-7
  modes, ... 7-6–7-7
  OUTPUT, ... 7-7
  structures, ... 7-4
  temporary, ... 15-27
FILLC procedure, ... 14-17
FILLSC procedure, ... 14-17
Flexible order of declarations, ... 9-3, 13-2
FLOAT function, ... 14-17
FLOAT4 function, ... 14-17
FOR statement, ... 12-15
FOR VAR feature, ... 12-17
Formal parameters, ... 11-9
Formatting
  parameters, ... 15-19
  rules, ... 15-15
FORTRAN
  attribute, ... 13-10
FORWARD directive, ... 13-5, 13-7
Forward pointer, ... 8-2
FREECT function, ... 14-17
Front end compiler errors, ... H-2
Functional
  parameter, ... 13-18
  types, ... 8-10
Functions
  A2DRQQ, ... 14-11
  A2SRQQ, ... 14-11
  ABS, ... 14-9
  ACDRQQ, ... 14-9

ACSRQQ, 14-9
AIDRQQ, 14-10
AISRQQ, 14-10
ALLHQQ, 14-10
ANDRQQ, 14-10
ANSRQQ, 14-10
ARCTAN, 14-10
arithmetic, 14-4
ASDRQQ, 14-10
ASSRQQ, 14-10
available, F-1
block, 13-1
BYLONG, 14-12
BYWORD, 14-12
CHDRQQ, 14-12
CHR, 14-12
CHSRQQ, 14-12
COS, 14-13
declarations, 13-1–13-3
DECODE, 14-13
designators, 1-8, 11-9
ENCODE, 14-15
EOF, 14-16, 15-2, 15-6
EOLN, 14-16, 15-2, 15-6
EXP, 14-17
FLOAT, 14-17
FLOAT4, 14-17
FREECT, 14-17
general discussion, 1-5
GTYUQQ, 14-18
heading, 8-10, 13-3, 13-7
HIBYTE, 14-18
HIWORD, 14-18
identifiers, 11-9, 13-3
LADDOK, 14-19
LDDRQQ, 14-19
LDSRQQ, 14-19
LMULOK, 14-19
LN, 14-19
LNDRQQ, 14-20
LNSRQQ, 14-20
LOBYTE, 14-20
LOCKED, 14-20

LOWER, ... 14-20
LOWORD, ... 14-20
MDDRQQ, ... 14-21
MDSRQQ, ... 14-21
MEMAVL, ... 14-21
MNDRQQ, ... 14-21
MNSRQQ, ... 14-21
MXDRQQ, ... 14-23
MXSRQQ, ... 14-23
ODD, ... 14-25
ORD, ... 14-25
parameters, ... 13-14
PIDRQQ, ... 14-26
PISRQQ, ... 14-26
POSITN, ... 14-26
PRED, ... 14-26
PRDPQQ, ... 14-26
PRSRQQ, ... 14-26
RESULT, ... 11-14, 14-29
RETYPE, ... 11-14, 14-29
ROUND, ... 14-31
ROUND4, ... 14-31
SADDOK, ... 14-31
SCANEQ, ... 14-31
SCANNE, ... 14-32
SHDRQQ, ... 14-32
SHSRQQ, ... 14-32
SIN, ... 14-32
SIZEOF, ... 14-32
SMULOK, ... 14-33
SNDRQQ, ... 14-33
SNSRQQ, ... 14-33
SQR, ... 14-33
SQRT, ... 14-33
SRDRQQ, ... 14-33
SRSRQQ, ... 14-33
SUCC, ... 14-33
THDRQQ, ... 14-33
THSRQQ, ... 14-33
TICS, ... 14-33
TNDRQQ, ... 14-34
TNSRQQ, ... 14-34
TRUNC, ... 14-34

TRUNC4, . . . 14-34
UADDOK, . . . 14-34
UMULOK, . . . 14-35
UPPER, . . . 14-36
WRD, . . . 14-36

G

GET procedure, . . . 14-18, 15-2–15-3
GOTO metacommand, . . . 17-6
GOTO statement, . . . 12-7
GTYUQQ function, . . . 14-18

H

Headings, . . . 8-10
HIBYTE function, . . . 14-18
HIWORD function, . . . 14-18

I

Identifiers
components, . . . 2-1
declaration, . . . 3-3
definition of, . . . 3-1
general discussion, . . . 1-12
predeclared, . . . 3-4
program, . . . 16-3
record field, . . . 6-13
restrictions, . . . 3-2
scope of, . . . 3-3
type, . . . 4-3
underscore, . . . Intro 3
IF
metacommand, . . . 17-9
statement, . . . 12-12
Implementation differences, . . . C-1
IN operator, . . . 11-8
INCLUDE metacommand, . . . 17-10
INCONST metacommand, . . . 17-10
INDEXCK metacommand, . . . 17-6
Indexed variable, . . . 10-5
INITCK metacommand, . . . 17-6

INPUT, 7-7, 16-4
INSERT procedure, 14-19
INTEGER type, 4-1, 9-5
INTEGER1 predeclared subrange, 5-5
INTEGER4
    errors, H-42
    type, 5-8, 9-5
Interactive files, Intro 3
Interface
    division, 16-13
    implementation, 16-14
Internal compiler errors, H-32
INTERRUPT attribute, Intro 3, 13-11
ISO Standard Pascal, Intro 2, 13-1
ISR operator, 11-5

L

Labels, 3-2, 12-2
LADDOK function, 14-19
Language levels, Intro 2
Lazy evaluation, Intro 3, 7-4, 15-7
LDDRQQ function, 14-19
LDSRQQ function, 14-19
Learning resources for Pascal, Intro 5
LEN notation, 10-5
Letters, 2-2
LINE metacommand, 17-7
LINESIZE metacommand, 17-12
LIST metacommand, 17-12
Listing file
    control, 17-12
    file format, 17-15
LMULOK function, 14-19
LN function, 14-19
LNDRQQ function, 14-20
LNSRQQ function, 14-20
LOBYTE function, 14-20
LOCKED function, 14-20
Loop label, 12-2, 12-17
LOWER function, 14-20
LOWORD function, 14-20

LSTRING
    super array type, ... 6-7, 9-7
    passing by reference, ... 4-4

M

M and N formatting parameters, ... 15-19
M parameter, ... 15-22
MARKAS procedure, ... 14-21
MATHCK metacommand, ... 17-7
MAXINT constant, ... 9-5
MAXINT4 constant, ... 9-5
MAXWORD constant, ... 9-5
MDDRQQ function, ... 14-21
MDSRQQ function, ... 14-21
MEMAVL function, ... 14-21
Memory errors, ... H-37
MESSAGE metacommand, ... 17-11
Messages, error, ... H-37
Metacommands
    available, ... 1-2
    $BRAVE, ... 17-5, 17-17
    $DEBUG, ... 17-5
    $ENTRY, ... 17-5
    $ERRORS, ... 17-6, 17-17
    $EXTEND, ... 17-3
    $GOTO, ... 17-6
    $IF/$THEN/$ELSE, ... 17-9
    $INCLUDE, ... 17-10
    $INCONST, ... 17-10
    $INDEXCK, ... 17-10
    $INITCK, ... 17-6
    $LINE, ... 17-7
    $LINESIZE, ... 17-12
    $LIST, ... 17-12
    $MATHCK, ... 17-7
    $MESSAGE, ... 17-11
    $NILCK, ... 17-7
    $OCODE, ... 17-13
    $PAGE, ... 17-13
    $PAGEIF, ... 17-13
    $PAGESIZE, ... 17-13

$POP, 17-11
$PUSH, 17-11
$RANGECK, 17-13
$ROM, 17-3
$RUNTIME, 17-13
$SIZE, 17-3
$SKIP, 17-13
$SPEED, 17-3
$STACKCK, 17-13
$SUBTITLE, 17-13
$SYMTAB, 17-13
$SYSTEM, 17-3
$TAGCK, 17-8
$TITLE, 17-14
$WARN, 17-8
notation, 17-1
optimization, 17-3
setting language level, 17-3
summary of, G-1
Metaconditional, 17-10
Metavariable, 17-1
MNDRQQ function, 14-21
MNSRQQ function, 14-21
MOD operator, 11-3
Modules, Intro 3, 16-6
MOVEL procedure, 14-22
MOVER procedure, 14-22
MOVESL procedure, 14-22
MOVESR procedure, 14-23
MS-Pascal
comparison with UCSD, C-4
comparison with other Pascals, C-1
file system errors, H-35
tutorial references, Intro 5
MXDRQQ function, 14-23
MXSRQQ function, 14-23

N

N and M formatting parameters, 15-19
N formatting parameter, 15-19
NaN (Not-a-Number), 11-7
NEW procedure, 14-23

NILCK metacommand, 17-7
Nondecimal numbering, Intro 3, 8-5, 9-6
NOT operator, 11-4
Not-a-Number (NaN) values, 11-7
Notation, syntactic groups, 1-13
NULL, 6-8
Null set, 6-18
Numeric constant, 9-3

O

OCODE metacommand, 17-13
ODD function, 14-25
Operands, 11-1
Operating system runtime errors, H-34
Operators
- *, 11-3, 11-8
- +, 11-3, 11-8
- −, 11-3, 11-8
- <, 11-7, 11-8
- <=, 11-7, 11-8
- < >, 11-7, 11-8
- =, 11-7, 11-8
- >, 11-7, 11-8
- >=, 11-7, 11-8
- ADR, 11-1
- ADS, 11-1
- AND, 11-4
- AND THEN, 12-19
- DIV, 11-4
- IN, 11-8
- ISR, 11-5
- MOD, 11-3
- NOT, 11-4
- OR, 11-4
- OR ELSE, 12-19
- precedence, 11-1, 11-11
- relational, 11-11
- SHL, 11-5
- SHR, 11-5
- XOR, 11-4

Optimization, 11-11, 12-4
OR ELSE operator, 12-19
OR operator, 11-4

ORD function, 14-25
Ordinal
  arithmetic errors, H-38
  types, 5-1
ORIGIN attribute, Intro 4, 10-11
OTHERWISE clause, Intro 4, 6-15, 12-13
OUTPUT, 7-7, 16-4
Ovals in syntax diagrams, A-1

P

PACK procedure, 14-25
PACKED
  types, 8-9
  limitations, 8-9
PAGE metacommand, 17-13
PAGE procedure, 14-26, 15-2, 15-7
PAGEIF metacommand, 17-13
PAGESIZE metacommand, 17-13
Panic errors, H-2
Parameters
  actual, 11-9, 13-15
  formal, 11-9, 13-15, 13-18
  function, 13-14
  functional, 13-18
  procedural, 13-18
  procedure, 13-18
  reference, 13-15
  value, 13-18
  VARS, 13-15
Pascal
  See ISO Standard Pascal
    MS-Pascal
    UCSD Pascal
Period (.), 1-3
PIDRQQ function, 14-26
PISRQQ function, 14-26
PLYUQQ procedure, 14-26
POP metacommand, 17-11
PORT attribute, 10-11
POSITN function, 14-26
PPMFQQ, 16-5
PPMUQQ, 16-5

Precision of REAL numbers, 9-4
PRED function, 14-26
Predeclared
- constants, 9-5
- functions, 14-1
- identifiers, 3-4
- procedures, 14-1
- types, 7-7, 8-5

Procedural
- parameter, 13-18
- types, 8-10

Procedures
- ABORT, 14-9
- ASSIGN, 14-10, 15-23
- available, F-1
- BEGOQQ, 14-11
- BEGXQQ, 14-11
- block, 13-1
- CLOSE, 14-12, 15-23
- CONCAT, 14-13
- COPYLST, 14-13
- COPYSTR, 14-13
- DATE, 14-13
- declarations 13-1
- DELETE, 14-17
- DISBIN, 14-14
- DISCARD, 14-14, 15-24
- DISPOSE, 14-14
- ENABIN, 14-15
- ENDOQQ, 14-15
- ENDXQQ, 14-16
- EVAL, 14-16
- FILLC, 14-17
- FILLSC, 14-17
- GET, 14-18, 15-2–15-3
- general discussion, 1-5
- heading, 8-10, 13-7
- INSERT, 14-19
- MARKAS, 14-21
- MOVEL, 14-22
- MOVER, 14-22
- MOVESL, 14-22
- MOVESR, 14-23

NEW, ... 14-23
PACK, ... 14-25
PAGE, ... 14-26, 15-2, 15-7
parameters, ... 13-18
PLYUQQ, ... 14-26
PTYUQQ, ... 14-27
PUT, ... 14-27, 15-2–15-3
READ, ... 14-27, 15-11, 15-13
READFN, ... 14-27, 15-24
READLN, ... 14-27, 15-11, 15-13
READSET, ... 14-27, 15-24
RELEAS, ... 14-27
RESET, ... 14-29, 15-2, 15-5
REWRITE, ... 14-30, 15-2, 15-5
SEEK, ... 14-32, 15-25
statement, ... 12-6
TIME, ... 14-34
UNLOCK, ... 14-35
UNPACK, ... 14-35
UPPER, ... 14-36
VECTIN, ... 14-36
WRITE, ... 14-37, 15-11, 15-18
WRITELN, ... 14-37, 15-11, 15-18
Programs
heading, ... 16-3
identifiers, ... 16-3
parameters, ... 16-3
PRSRQQ function, ... 14-27
PSDRQQ function, ... 14-27
PTYUQQ procedure, ... 14-27
PUBLIC attribute, ... Intro 3, 10-10
Punctuation, ... 2-3
PURE attribute, ... 13-12
PUSH metacommand, ... 17-11
PUT procedure, ... 14-27, 15-2–15-3

Q

Question mark (?), ... 2-6
QQ naming convention, ... 3-2

R

R address notation, 8-5, 8-8
Radix separator, 9-6
Random access files, Intro 3
RANGECK metacommand, 17-8
READ
- formats, 15-15
- procedure, 14-27, 15-11, 15-13

READFN procedure, 14-27, 15-24
READLN procedure, 14-28, 15-11, 15-13
READONLY attribute, 10-12
READSET procedure, 14-28, 15-24
REAL
- arithmetic errors, H-40
- constants, 9-4
- precision, 9-4
- type, 5-6

Records
- field offsets, 6-16
- fields, 6-13
- variables, 10-5, 12-18

Recursion, 8-2, 10-9
Reference
- parameters, 13-15
- variables, 10-6

Relational operator, 11-6
RELEAS procedure, 14-28
REPEAT statement, 12-14
Repetition statement, 12-14
Reserved words, 2-5, E-1
RESET procedure, 14-29, 15-2, 15-5
RESULT function, 11-14, 13-4, 14-29
RETURN statement, Intro 4, 12-9
RETYPE function, 11-14, 14-29
REWRITE procedure, 14-30, 15-5
ROM metacommand, 17-3
ROUND function, 14-31
ROUND4 function, 14-31
Runtime
- errors, H-37
- file system errors, H-32

RUNTIME metacommand, 17-8

S

S address notation, 8-5, 15-16
SADDOK function, 14-31
SCANEQ function, 14-31
SCANNE function, 14-32
SEEK procedure, 14-32, 15-25
Segment parameters, 8-6, 13-17
Semaphores, 14-8
Separators, 2-2, 2-5
SEQUENTIAL, 7-7
Sequential control, 12-19
Sets
- constructors, 11-8
- expressions, 11-8
- types, variables, 6-18

SHDRQQ function, 14-32
SHL operator, 11-5
SHR operator, 11-5
SHSRQQ function, 14-32
Simple statement, 12-4
SIN function, 14-32
SINT predeclared subrange, 5-5
SIZE metacommand, 17-3
SIZEOF function, 14-32
SKIP metacommand, 17-13
SMULOCK function, 14-33
SNDRQQ function, 14-33
SNSRQQ function, 14-33
SPEED metacommand, 17-3
SQR function, 14-33
SQRT function, 14-33
Square brackets, 11-8
SRDRQQ function, 14-33
SRSRQQ function, 14-33
STACKCK metacommand, 17-8
Standard Pascal
  See ISO Standard Pascal
Statements
- assignment, 12-4
- BREAK, Intro 4, 12-9, 12-17
- CASE, 12-13
- compound, 12-10

conditional, 12-11
CYCLE, Intro 4, 12-9, 12-17
FOR, 12-15
general discussion, 1-6
GOTO, 12-7
IF, 12-12
procedure, 12-6
REPEAT, 12-14
repetition, 12-14
RETURN, Intro 4, 12-9
sequential control, 12-19
simple, 12-1, 12-4
structured, 12-1, 12-10
WHILE, 12-14
WITH, 12-18
STATIC attribute, Intro 4, 10-9
String
constant, 9-7
read, Intro 3
super array type, 14-6
Structured
constants, 1-10, Intro 3, 1-10, 9-8
statements, 12-10
type errors, H-42
types, 6-1
Subrange
predeclared, 5-5
types, 5-4
SUBTITLE metacommand, 17-13
SUCC function, 14-33
Super array
features, Intro 3
parameters, 13-16
type, 14-25
super type, 6-3
Super types, 4-3
Switches, 17-18
SYMTAB metacommand, 17-13
Syntax diagrams, A-1
System level
intrinsics, 14-6
Pascal, Intro-3
SYSTEM metacommand, 17-3

T

Tag field, ... 6-14
TAGCK metacommand, ... 17-8
Temporary files, ... 15-27
Terminal I/O routines, ... 14-8
TERMINAL mode, ... 7-6
Textfile I/O, ... 15-11
THDRQQ function, ... 14-33
THSRQQ function, ... 14-33
TICS function, ... 14-33
TIME procedure, ... 14-34
TITLE metacommand, ... 17-14
TNDRQQ function, ... 14-34
TNSRQQ function, ... 14-34
TRUNC function, ... 14-34
TRUNC4 function, ... 14-34
Tutorial list for Pascal, ... Intro-5
Type identity, ... 4-4
Types
- ARRAY, ... 6-2
- base, ... 6-18
- BOOLEAN, ... 5-3
- CHAR, ... 5-3, 9-7
- compatibility, ... 4-5
- components, ... 6-2
- declaration, ... 4-3
- definition, ... 4-1
- enumerated, ... 5-3
- FILE, ... 8-1
- functional, ... 8-10
- general discussion, ... 1-11
- identifiers, ... 4-3
- INTEGER, ... 4-1, 9-5
- INTEGER4, ... 5-8, 9-5
- LSTRING, ... 6-7, 9-7
- ordinal, ... 5-1
- PACKED, ... 8-9
- pointer, ... 8-2
- procedural, ... 8-10
- REAL, ... 5-6, 9-4
- records, ... 6-13
- reference, ... 8-1
- reference parameters, ... 4-4

sets, ... 6-18
simple, ... 5-1, 11-2
STRING, ... 6-6, 9-7
structured, ... 6-1
subrange, ... 5-4
super arrays, ... 6-3
WORD, ... 5-2, 9-5

U

UADDOK function, ... 14-34
UCSD Pascal, ... C-4
UMULOK function, ... 14-35
Underscore character, ... Intro 3, 2-2
Units,
constituents, ... 16-10
identifiers, ... 16-10
implementation, ... 16-8, 16-14
interfaces, ... 16-8, 16-13
separate compilation, ... Intro 4
UNLOCK procedure, ... 14-35
UNPACK procedure, ... 14-35
Unused characters, ... 2-6
Upper bound, ... 6-3, 6-4
UPPER function, ... 14-36

V

Value parameters, ... 13-18
VALUE section, ... Intro 3, 1-9, 10-3
VAR
in FOR statement, ... 12-17
parameters, ... 10-2
sections, ... 10-2
Variable length strings, ... Intro 3
Variables
attributes of, ... 10-8
declarations, ... 10-2
general discussion, ... 1-9
initial values, ... 13-3
reference, ... 10-6
values, ... 10-1

Variant records, ... 6-14
VARS parameters, ... 8-6, 13-15
VECTIN procedure, ... 13-13, 14-36

W

WARN metacommand, ... 17-8
WHILE statement, ... 12-14
WITH statement, ... 12-18
WORD type, ... Intro 3, 5-2, 9-5, 11-2
WRD function, ... 14-36
WRITE
    formats, ... 15-19
    procedure, ... 14-36, 15-11, 15-18
WRITELN procedure, ... 14-36, 15-11, 15-18

X

XOR operator, ... 11-4