# *SPARCompiler Pascal 3.0.2*

*User's Guide*

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

Part No.: 801-5054-10
Revision A, December 1993

# *Contents*

# *Figures*

# *Tables*

# *Preface*

## *Purpose*

This manual describes the SPARCompiler Pascal product running in the Solaris environment. The purpose of this guide is to help you begin writing and compiling Pascal programs on a Sun workstation. Many of the examples and references in this guide are based on the SPARCompiler Pascal 3.0.2 product that runs on the Solaris 2.2 environment or above. For other release-specific information, please read the Product Notes and the `README` files that accompany each release of the Pascal product.

## *Operating Environment*

Solaris 2.2 implies:

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

## *Installation*

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

## *Audience*

This guide was prepared for software engineers who write Pascal programs with a Sun™ workstation. This guide assumes you are familiar with ISO standard Pascal and the SunOS™ operating system. This guide contains an introduction to the SunOS file system and describes how complex programs are built in this system.

## *Organization*

This guide contains the following chapters and sections:

- Chapter 1, “Introduction,” gives basic information about the Pascal compiler and related program development tools.
- Chapter 2, “Using Pascal,” describes how to write, compile, and run a simple Pascal program.
- Chapter 3, “Using pc, the Pascal Compiler,” describes the `pc` command and its options.
- Chapter 4, “The File System,” is an introduction to the file system.
- Chapter 5, “Program Construction and Management,” is an introduction to how complex programs are built in Pascal.
- Chapter 6, “Separate Compilation,” describes how programs can be divided into several units, compiled, and linked. (Chapter 5 covers much of this information at a more introductory level.)
- Chapter 7, “The C–Pascal Interface,” describes how to write programs that are partly in C and partly in Pascal.
- Chapter 8, “The FORTRAN–Pascal Interface,” describes how to write programs that are partly in FORTRAN and partly in Pascal.
- Chapter 9, “Error Diagnostics,” describes the error diagnostics of Pascal.
- Chapter 10, “Using the XView Toolkit,” describes how to use the XView toolkit with Pascal.
- Appendix A, “Pascal Preprocessor,” describes the Pascal preprocessors, concentrating on the nonstandard preprocessor `cppas`.
- Appendix B, “Compiler Error Messages,” lists all error messages the compiler produces.

## *Conventions Used in This Guide*

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

*Table 0-1* BNF Meta Symbols

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

This guide also uses the following conventions:

**`Bold face typewriter font`**
Indicates commands that you should type in exactly as printed in the guide.

`Regular typewriter font`
Represents what the system prints on your workstation screen, as well as Pascal keywords, identifiers, program names, and file names.

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

`hostname%`
Represents your system prompt for a nonprivileged user account.

Screen box
Encloses a program listing or text that represents an interactive session. In the interactive examples, user input is indicated by **`boldface typewriter font`**, while information printed by the computer is indicated by `regular typewriter font`. Variable information is indicated by *italic font*. All appear in a reduced size.

## *Related Documentation*

This guide was designed to accompany the Sun manual *Pascal 3.0.2 Reference Manual*. The *Standard Pascal User Reference Manual* describes the ISO Pascal Standard. The *Pascal 3.0.2 Reference Manual* describes extensions to Standard Pascal. You may find it helpful to have those manuals at hand while you are reading through this manual.

Pascal also provide manual pages, or on-line documentation, on Pascal commands. For example, pc(1) describes the Pascal compiler. The on-line documentation is included in the Pascal package and must be installed with the rest of the software. Once you install the documentation, you can read about pc by entering the following command:

```
hostname% man pc
```

Other reference material include the following Sun manuals:

*Profiling Tools*
*Numerical Computation Guide*

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

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

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

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

# *Introduction* 1

## *Standards*

This manual describes the SPARCompiler Pascal product running in the Solaris environment. SPARCompiler Pascal is a derivative of the Berkeley Pascal system distributed with UNIX[1] 4.2 BSD. It complies with FIPS PUB 109 ANSI/IEEE 770 X3.97-1983 and BS6192/ISO7185 at both level 0 and level 1.

## *Pascal Compiler*

The name of the Pascal compiler is `pc`. If given an argument file ending with `.p` or `.pas`, `pc` compiles the file and leaves the result in an executable file, called `a.out` by default.

## *Features*

The SPARCompiler Pascal includes many extensions to the standard, including the following:

- Separate compilation of programs and modules
- `dbx` (symbolic debugger) support
- Optimizer support

1. UNIX is a registered trademark of UNIX Systems Laboratory.

# 1

- Multiple `label`, `const`, `type`, and `var` declarations
- Variable-length character strings
- Compile-time initializations
- `static` and `extern` declarations
- Different sizes of integer and real data types
- Integer constants in any base from 2 to 16
- Extended input/output facilities
- Extended library of built-in functions and procedures
- Universal and function and procedure pointer types
- Direction of parameter passing: into a routine, out of a routine, or both

---

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

---

## *Compatibility (SPARCompiler Pascal 3.0.2)*

In general, SPARCompiler Pascal 3.0.2 runs in the Solaris® 2.2 operating environment. For this release Solaris® 2.2 implies:

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

---

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

---

### *Pascal 3.0.2 Incompatibility with Pascal 2.1 Binaries*

Libraries and `.o` files compiled with Pascal 2.1 or earlier are not compatible with Pascal 3.0.2.

## *Text Editors*

The operating system provides two main editors.

textedit
: The window-based text editor, `textedit`, runs in the OpenWindows™ window environment.

vi
: The standard visual display editor, `vi`, offers the capabilities of both a line and a screen editor. It also provides several commands for editing programs. For example, the `autoindent` option provides white space at the beginning of a line, and the `showmatch` option shows matching parentheses. For more information, see the *Editing Text Files* manual.

## *Debuggers*

The SPARCworks product supplies two tools for debugging your Pascal program. SPARCworks is optionally available with Pascal.

dbx
: A symbolic debugger.

`debugger`
: A window- and mouse-based version of the symbolic debugger, which runs in the OpenWindows window environment.

## *SPARCworks Source Browser*

SPARCworks SourceBrowser™ is a source code browser that makes it easy to find occurrences of any symbol in your program. It can find them in all source files, including header files.

SPARCworks is optionally available with Pascal.

You can run SourceBrowser in either a window or command-line environment as follows:

sbrowser
: OpenWindows interfaces to SourceBrowser. See *Browsing Source Code* in the SPARCworks manual set for a complete description of `sbrowser`.

sbquery
The command-line interface to SourceBrowser. See the `sbquery`(1) manual page for details.

The SourceBrowser also gives you access to the CallGrapher, which lets you graphically identify the interrelationships of the functions in your program. For more information on the CallGrapher, see the `sbrowser`(1) manual page and the *Browsing Source Code* manual.

You can also use the Analyzer, which is a graphical data analysis tool that helps analyze performance data that is collected using the Collector in dbxtool. In addition to analyzing the data, the Analyzer also provides a reordering functionality. For more information on the Analyzer, see the `analyzer`(1) manual page.

## *XView Toolkit*

The XView Application Programmer's Interface is an object-oriented, server-based, user-interface toolkit for the X Window System Version 11 (X11). It is designed to help programmers manipulate XView windows and other XView objects.

## *Native Language Support*

SunPro supports the development of applications in languages other than English. This includes most European languages and Japanese. As a result, you can easily switch your application from one native language to another. This is known as *internationalization*.

### *Internationalization*

A product can support up to four levels of internationalization:

Level 1
Allows native-language characters (such as the a-umlaut). This is referred to as the 8-bit clean model because the eighth bit in each byte is used to represent native-language characters.

Level 2
: Recognizes and displays international date and time formats (such as 26.07.90 in Italian); international decimal units (such as 1.234.567,89 in French); and international monetary units (such as 1.234,56 Pts in Spanish).

Level 3
: Contains support for localized messages and text presentation.

Level 4
: Contains Asian language support.

The Pascal compiler currently supports Level 1 (it allows native language characters in comments, strings, and data) and most of Level 2 (it time-stamps program listings with international date and time formats). However, the compiler does not allow input and output in the various international formats. If it did, it would not comply with the Pascal language standard, ANSI/IEEE 770 X3.97-1983.

For example, the standard specifies a period (.) as the decimal unit in the floating-point representation. Consider the following program, which prints a floating-point value.

```
program sample(output);

var r : real := 1.2;

begin
     writeln(r);
end.
```

When you compile and run the program on the internationalized Pascal compiler, you get:

```
1.20000000000000e+00
```

If you reset your system locale to, for example, France and rerun the program, you still get the same output. The period won't be replaced with a comma, the French decimal unit.

# 1

## Locale

You can change your application from one native language to another by setting the locale. For information on this and other native language support features, see the *Solaris 2.2 Developer's Guide to Internationalization*.

# Manuals

Other manuals containing information on editing or using Pascal and the operating system are:

- *Pascal 3.0.2 Reference Manual*
- *Numerical Computation Guide*
- *Profiling Tools*
- *Installing SunPro Software on Solaris*
- *Browsing Source Code*
- *Debugging a Program*
- *Performance Tuning an Application*

## Documents in Hard Copy and in the AnswerBook System

The following documents are on-line and in hard copy, as shown:

*Table 1-1* Documents in Hard Copy and in the AnswerBook System

| **Title** | **Hard Copy** | **On-line** |
|---|---|---|
| SPARCompiler Pascal User's Guide | X | X |
| SPARCompiler Pascal Reference Manual | X | X |
| Installing SunPro Software on Solaris | X | X |
| Profiling Tools | | X |
| Product Notes | X | |

### *Other Reference Material*

There is also a floating-point white paper called *"What Every Computer Scientist Should Know About Floating-Point Arithmetic,"* by David Goldberg. This is included in `/opt/SUNWspro/READMEs` directory and is called `floating-point.ps`.

The `floating-point.ps` file is in PostScript and can be printed on any PostScript-compatible printer that has Palatino font. It can be viewed on-line by using the `pageview` command:

```
% pageview floating-point.ps
```

The on-line `README` directory is described in the *Product Notes*.

## *Licensing*

This compiler uses network licensing, as described in the manual *Installing SunPro Software on Solaris*.

When you invoke the compiler, if a license is available, the compiler simply starts. If no license is available, your request for a license is put on a queue, and your compile continues when a license becomes available. A single license can be used for any number of simultaneous compiles by a single user on a single machine. There are two licensing related options:

- `-noqueue` = do not queue request if no license is available
- `-xlicinfo` = return information on the status of licensing

The `-xlicinfo` option does not check out a license.

For details on getting a license (where to call, what information to give them), read *Installing SunPro Software on Solaris.*

# 1

# *Using Pascal* 2

This chapter provides essentials about compiling and executing a Pascal program.

Using Pascal requires three steps:

- Writing a program in Pascal using an editor and giving the file name a `.p` (or `.pas`) suffix
- Compiling the `.p` (or `.pas`) file using the `pc` command
- Executing the program by typing the name of the executable file

## *A Simple Pascal Program*

The following is a simple Pascal program that converts temperatures from Fahrenheit to Celsius. Use an editor to enter this program on your system and store it in a file called `temp.p`.

```
program temperature(output) ;

{ Program to convert temperatures from
 Fahrenheit to Celsius. }

const
    MIN = 32 ;
    MAX = 50 ;
    CONVERT = 5 / 9 ;

var
    fahren: integer ;
    celsius: real ;

begin
    writeln('Fahrenheit       Celsius') ;
    writeln('----------       -------') ;
    for fahren := MIN to MAX do begin
         celsius := CONVERT * (fahren - 32) ;
         writeln(fahren: 5, celsius: 18: 2) ;
 end ;
end.
```

## *Compiling the Program*

Now compile the program with `pc`, which is the Pascal compiler.

```
hostname% pc temp.p
```

Pascal names the compiled version of the program `a.out` by default.

## *Running the Program*

To run the program, simply enter `a.out`. Here is the output of `temp.p`.

```
hostname% a.out
  Fahrenheit Celsius
  -----------------
    32                0.00
    33                0.56
    34                1.11
    35                1.67
    36                2.22
    37                2.78
    38                3.33
    39                3.89
    40                4.44
    41                5.00
    42                5.56
    43                6.11
    44                6.67
    45                7.22
    46                7.78
    47                8.33
    48                8.89
    49                9.44
    50                10.00
hostname%
```

## *Renaming the Executable File*

It is inconvenient to have the result of every compilation in a file called `a.out`. If such a file already exists, it is overwritten. You can avoid this in either of the two following ways:

- Change the name of `a.out` after each compilation with the `mv`(1) command:

```
hostname% mv a.out temp
```

- Use the compiler `–o` option to name the output executable file. This example places the executable code in the file `temp:`

```
hostname% pc –o temp temp.p
```

Now run the program by typing the name of the executable file:

```
hostname% temp
Fahrenheit Celsius
---------- -------
   32           0.00
   33           0.56
   34           1.11
   .            .
   .            .
   .            .
```

## A Simple Interactive Program

In Pascal, the predefined file variable `input` is equivalent to the operating system standard input file `stdin`. Similarly, the file variable `output` is equivalent to the standard output file `stdout`.

Following is a simple Pascal program that copies input to output. Use an editor to enter this program on your system and store it in a file called `copy.p`:

```
program copy(input, output);

{ This program copies input to output. }

var
    c: char;

begin
    while not eof do begin
      while not eoln do begin
          read(c);
          write(c)
      end;
      readln;
      writeln
    end
end. { copy }
```

## *Compiling the Program*

Use the `pc` command to compile the program and store it in the executable file `copy`. Here is the command format:

```
hostname% pc -o copy copy.p
```

## *Running the Program*

Because the standard files `input` and `output` default to the terminal, the program simply echoes each line you type. The program terminates when you type the end-of-file (Control-D) character at the beginning of a line. Try entering the following:

```
hostname% copy
hello, are you listening?
hello, are you listening?
goodbye, I must go now.
goodbye, I must go now.
 (CTRL-D)
 hostname%
```

## *Redirecting I/O*

To write the output to a file instead of to the terminal, use the redirection operator, `>`, followed by a file name. For instance, to write to a file called `data`, enter the following:

```
hostname% copy > data
 hello, are you listening?
 goodbye, I must go now.
 (CTRL-D)
 hostname%
```

Using the same program, but with the `<` operator to redirect input, you can print the file on the terminal:

```
hostname% copy < data
 hello, are you listening?
 goodbye, I must go now.
 hostname%
```

## *Using a File Name as a File Variable*

You can also redirect the output by listing the file as a file variable in the program statement. The Pascal library associates the file variable with a file of the same name. For example, `copy2.p` lists `data` as the input file variable:

```
program copy2(data, output);

{ This program redirects input. }

var
    c: char;
    data: text;

begin
    reset(data);
    while not eof(data) do begin
      while not eoln(data) do begin
          read(data, c);
          write(c)
      end;
      readln(data);
      writeln
    end
end. { copy2 }
```

Assuming that the file `data` is still in the current directory, you can compile and run the program as follows:

```
 hostname% pc -o copy2 copy2.p
 hostname% copy2
 hello, are you listening?
 goodbye, I must go now.
 hostname%
```

# Where Did My Program Fail?

Pascal makes it easy to understand a program that failed. The compiler provides a traceback utility that finds the routine that triggered the error.

## Pascal Traceback

The Pascal traceback installs signal handlers on selected signals and dumps a backtrace when those signals are caught. The backtrace shows the chain of calls leading from the routine in which the error occurred, all the way back to the main program.

The set of signals Pascal catches is `SIGQUIT`, `SIGILL`, `SIGTRAP`, `SIGIOT`, `SIGABRT`, `SIGEMT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, `SIGPIPE`, `SIGTERM`, and `SIGLOST`. See the `signal`(3) man page for further information on these signals.

After the system produces the traceback, it continues with whatever action it would have done had the interposer not been in place, including calling a previously set user signal handler.

The traceback facility uses the debugger `dbx`. In order to get a traceback, SPARCworks must be installed on your system and the directory containing `dbx` must be in your PATH environment variable. If the traceback routine cannot find `dbx`, it will not produce the traceback.

## A Simple Program (with Segmentation Violation)

A segmentation violation occurs when your program tries to reference memory outside your address space. The operating system catches this and generates an error message. Following is an example program, sample.p, which contains a segmentation violation:

```
program segmentationViolation;
type p = ^integer;
var null_ptr : p;

  procedure error_in_here(arg1:integer; arg2:char);
  begin arg2:='B';
    null_ptr := nil;
    { next statement causes a SEGV }
    arg1 := null_ptr^;
  end;

procedure call1(my_int : integer; my_other_int:integer);
  procedure nested_proc(n_int:integer);
      begin
         n_int:=22;
         error_in_here(77,'c');
      end;
  begin
    my_int:=11; my_other_int:=33;
    nested_proc(2);
  end;

begin
  call1(55, 66);
end.
```

## *Compiling and Running the Program*

When you compile and run the program, you should receive output similar to the following in which the first line indicates the name of the offending signal (in this case, a segmentation violation):

```
hostname % pc segviol.p
hostname % a.out

*** a.out terminated by signal 11: bus error
*** Traceback being written to a.out.trace
Abort (core dumped)

hostname % more a.out.trace
*** Stacktrace of a.out
*** Program terminated due to bus error
=>[1] waitid(0x0, 0x28da, 0xeffff398, 0x3, 0x3, 0x0), at 0xef70a008
  [2] _waitpid(0x28da, 0xeffff4ec, 0x3, 0x7efefeff, 0x81010100, 0x0), at 0xef725008
  [3] __PC0__sigdie(0x1130c, 0x1130c, 0xeffff500, 0xeffff560, 0xeffff500, 0xeffff53c), at
0xef77f19c
---- called from signal handler with signal 11 (SIGSEGV) -----
  [4] error_in_here(), at 0x1130c
  [5] nested_proc(0x2, 0xeffff930, 0x0, 0xef77ed58, 0xb, 0x21), at 0x11344
  [6] call1(0x37, 0x42, 0x2a1d8, 0x2a2d8, 0x2a2d8, 0x0), at 0x11388
  [7] program(0x2a1d8, 0x2a1e4, 0x2a2d4, 0x2a2d8, 0xef79143c, 0x40), at 0x113c4
  [8] main(0x1, 0xeffffa64, 0xeffffa6c, 0x2a000, 0x0, 0x0), at 0xef77ce18
detaching from process 10457
```

In this example, `error_in_here` reported the error. The `error_in_here` function was called by `call1.nested_proc`, which was called by `call1`, which was called by the main program. Routine names such as `call1.nested_proc` indicate a nested routine. If Pascal cannot find the name of a routine (for example, because the executable file has been stripped), it prints the hex address.

### *Using the* `-g` *Option*

If you compile the program with the `-g` option, the traceback also reports arguments, line number, and file name of each routine. Try compiling `segviol.p` with `-g`:

```
hostname % pc -g segviol.p
hostname % a.out

*** a.out terminated by signal 11: bus error
*** Traceback being written to a.out.trace
Abort (core dumped)

hostname % more a.out.trace

*** Stacktrace of a.out
*** Program terminated due to bus error
=>[1] waitid(0x0, 0x300e, 0xeffff398, 0x3, 0x3, 0x0), at 0xef70a008
  [2] _waitpid(0x300e, 0xeffff4ec, 0x3, 0x7efefeff, 0x81010100, 0x0), at 0xef725008
  [3] __PC0__sigdie(0x1130c, 0x1130c, 0xeffff500, 0xeffff560, 0xeffff500, 0xeffff53c), at
0xef77f19c
  ---- called from signal handler with signal 11 (SIGSEGV) ------
  [4] error_in_here(arg1 = 77, arg2 = 'B'), line 12 in "segv.p"
  [5] nested_proc(n_int = 22), line 19 in "segv.p"
  [6] call1(my_int = 11, my_other_int = 33), line 25 in "segv.p"
  [7] program(), line 29 in "segv.p"
detaching from process 12298
```

The program prints the ASCII values of character variables.

If you compile some modules with `-g` and others without, the line numbers may not be accurate for all routines.

## *Disabling Traceback*

Use the `-notrace` command line option to disable traceback.

# *Using* pc, *the Pascal Compiler* 3

The name of the Pascal compiler is pc. If you give pc as an argument a filename that ends with .p (or .pas), pc compiles the file and leaves the result in an executable file, called a.out by default.

The format of this command follows.

pc [*options*] *filename*

## pc *Version Number*

To identify the version number of pc when you compile your program, call the compiler with the –V option. This option tells the compiler to produce a listing of the program. The version appears in the first line of the listing as follows:

```
hostname % pc -V hello.p -c
pc: SC3.0 18 Dec 1993
cpp: SC3.0 18 Dec 1993
pc0: SC3.0 18 Dec 1993 Pascal 3.0.2
fbe: C Development Set  (CDS) SPARCompilers 2.0.1 03 Sep 1992
pc3: SC3.0 18 Dec 1993
ld: Software Generation Utilities (SGU) SunOS/SVR4 (LK-1.1)
```

To identify the version number given an executable or object file created by the Pascal compiler, use the following command:

```
hostname % mcs -p a.out | grep Pascal
SC3.0 18 Dec 1993 Pascal 3.0.2
```

## *Compile/Link Sequence*

You can compile the file `any.p` with the following command. line:

```
hostname% pc any.p
```

This command actually invokes the compiler driver, which calls several programs or passes the program, each of which processes the program. The output of each pass is the input to the next one.

After several passes, the object file `any.o` is generated. An executable file then is generated with the default name `a.out`. Finally, the file `any.o` is removed.

The Pascal compiler, `pc`, does the following:

- Calls `cpp`, the C preprocessor or `cppas`, the preprocessor used when you use the `-xl` option
- Calls `pc0`, the Pascal front end
- Calls the global optimizer if you use the `-O` option
- Calls `cg`, the code generator
- Calls `fbe`, the assembler, which generates the relocatable object file
- Calls `pc3`, which checks for symbol name conflicts
- Calls `ld`, the linker / loader, which generates the executable files using any libraries necessary to resolve undefined symbols

This is the default action of `pc`; some compiler options change what `pc` calls.

Figure 3-1 shows the sequence of events when you invoke pc.

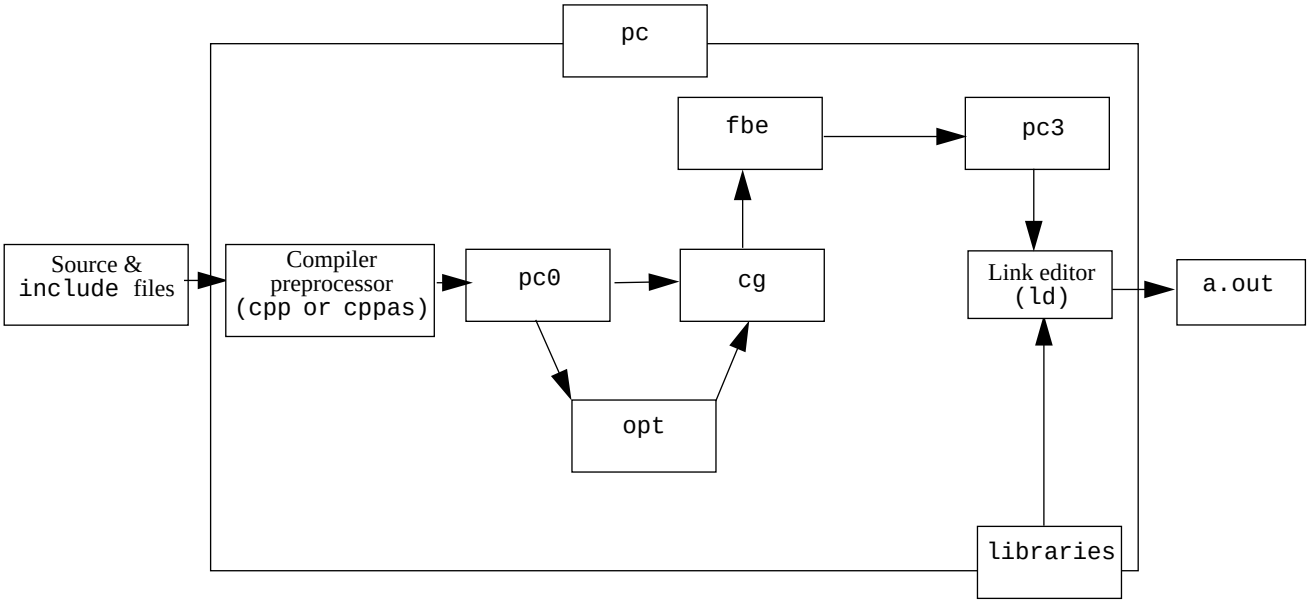


*Figure 3-1* Organization of Pascal Compilation

## *Language Preprocessor*

The cpp(1) program is the C language preprocessor. The Pascal compiler driver pc normally calls cpp(1) during the first pass of a Pascal compilation. If you use the –xl switch, pc calls the alternate preprocessor cppas. Then cpp(1) and cppas will operate on files having the extension .p or .pas.

You can give directives to cpp(1) or cppas to define constants, conditionally compile parts of your program, include external files, and take other actions. For example, the following program shows the use of an include directive, which asks cpp(1) to copy the named file into the program before compilation.

```
program showinclude;
#include "file.i"
begin
...
end.
```

See the man page for cpp(1) for information on its directives and other features. Appendix A, “Pascal Preprocessor documents cppas.

## *File name Extensions That Are Accepted by* pc

Pascal source files generally use the extension .p. The compiler recognizes other file name extensions. Table 3-1 lists the most important extensions.

The table notes that pc can produce assembler source files as well as unlinked object files. In each case, you can pass these partially compiled files to pc and it will finish compilation, linking, and loading.

*Table 3-1* File name Suffixes Pascal Recognizes

| Suffix | Description |
|---|---|
| .p | Usual extension for Pascal source files. |
| .pas | Valid extension for a Pascal source file. The extension tells pc to put object files in the current directory. The default name of the object file is the name of the source file but with a .o suffix. |
| .pi | Default extension for Pascal source files that have been processed by the Pascal preprocessor (either cpp or cppas). |
| .s | Extension for assembler source files that are produced when you call pc with the -S option. |
| .o | Extension for object files that are generated by the compiler when you call pc with the -c option. |

## Passing Options on the Command Line

To pass an option on the command line, use a dash (-) followed by the option name. In some cases, you will need to supply an additional piece of information, such as a file name. For example, this command enables the listing option `-l`, which is off by default.

```
hostname% pc -l rmc.p
```

This command causes the generated object file to be `rmc` instead of the default `a.out`.

```
hostname% pc -o rmc rmc.p
```

## Passing Options in the Program Text

Some options can be passed to the compiler in program text as well as on the command line. This facility lets you use different option values for different parts of a program.

Here are four examples of how options can be passed in program text:

```
{$P+}
{$H*}
(*$I-*)
{$l+,L-,n+}
```

The options that can be passed in program text are shown in Table 3-2:

*Table 3-2* Options That Can Be Passed in Program Text

| Option | Description |
|---|---|
| `b` | Buffering of the file `output`. |
| `C` | Runtime checks (same as `t`). |
| `calign` | Use C data formats. |
| `H` | Check heap pointers. |
| `l` | Make a listing. |
| `L` | Map identifiers and keywords to lowercase. |

| | |
|---|---|
| p | Statement limit counting (different from command-line p[1]). See stlimit in the *Pascal 3.0.2 Reference Manual*. |
| P | Partial evaluation of boolean expressions. |
| t | Runtime checks (same as C) (different from command-line t[1]). |
| u | Trim to 72 character line (not usable on command-line). |
| w | Print warning diagnostics. |

You set options within comments, and the comments can be delimited by either { and } or (* and *). The first character in the comment must be the $ (dollar sign). The $ must be immediately followed by an option letter. Next must appear either +, -, or *. (The option b is a special case which is addressed in "The Option b" on page 25.)

If you want to set more than one option, you can next have a comma, followed by another option letter and +, -, or *. You can set any number of options on a single line. There must be no embedded spaces. You can place spaces and other ordinary comment text after the last +, -, or *.

The new option setting takes effect at the next noncomment token.

The symbols + (plus sign) and - (minus sign) turn the option on and off, respectively. To understand the symbol *, you need to know something of how options work in this version of Pascal.

Except for b, each option in Table 3-2 has a current value and a "first in, last out" stack of up to 16 values. Again except for b, each option can be on or off.

When you begin compiling a program, each stack is empty and each option has an initial current value, which may be the option default value or may have been set on the command line.

1. The options p and t are different when they are used within program text and when they are used on the command line. This is because they are received directly by pc0 when they are used in program text, while the compiler driver gives them to other compiler passes when they are given on the command line. If you want to set them on the command line and also want them to have the same effect as passing them in program text, you can use the Qoption command-line option to pass them directly to pc0.

When the compiler encounters an option followed by a +, the current value is pushed onto the stack and the current value becomes ON. When the compiler encounters an option followed by a -, the current value is pushed onto the stack and the current value becomes OFF. When the compiler encounters an option followed by a *, the last value is popped off the stack and becomes the current value. (If no values have been pushed onto the stack, the effect of * is undefined.)

Figure 3-2 illustrates how options are passed in program text by showing a program and its output displayed next to one another.

```
program options (output);
begin
{$l+ Turns on listing}
     writeln (‘After $l-’);
{$l- Turns off listing}
{Notice that this line prints.}
     writeln (‘After $l+’);
{$l* Turns listing on again}
{Notice that this line does not print.}
    writeln (‘After $l*’)
end.
```

```
demo% pc options.p
Wed Aug 21 17:33:18 1991 options.p:
 4  writeln (‘After $l-’);
 5 {$l- Turns off listing}
 6 {Notice that this line prints.}
 10 writeln (‘After $l*’)
 11 end.
demo%
```

*Figure 3-2* Setting Listing Option in Program Text

## *The Option* b

Unlike the other options that can be set within programs, b has three settings: 0, 1, and 2. Use these numbers in place of +, -, and *. The option b has no stack; hence the * option cannot be used with b. You can only use this option in the main program. The block buffering value in effect at the end of the main program is used for the entire program.

## *A Note About the* caligh *Option*

The caligh option only makes sense in the type or var section of a program. Types or variables that are defined when caligh is turned on are stored with C-style formats for the entire program.

# 3

## *Frequently Used Options*

This section describes the seven most frequently used options. See section  on page 29 for descriptions of all options.

### *Suppress Linking:* `-c`

The `-c` options tells the compiler not to call the linker, `ld`(1). The Pascal compiler, `pc`, leaves a `.o` file for each source file. Without `-c`, `pc` calls the linker when it finishes compilation.

### *Define Name to Preprocessor:* `-D`*name*`[=`*def*`]`

The `-D` option defines a symbol *name* to the C preprocessor, `/usr/ccs/lib/cpp`. It is equivalent to using the `#define` statement in your program. If you do not include a definition, *name* is defined as 1. See `cpp`(1) for more information.

### *Debugging Information:* `-g`

The `-g` option[1] tells `pc` to produce additional symbol table information for `dbx` and `debugger`.

#### *Behavior Change: Debugging Optimized Code*

You can now compile using both the `-g` and `-O` options. However, there are some side effects:

- The `next` and `step` commands do not work, but the `cont` command does.
- If you have make files that rely on `-g` overriding `-O`, you will have to revise those files.
- If you have make files that check for a warning message that `-g` overrides `-O`, you must revise those make files.

1. In earlier versions of this Pascal, you could not use the `-g` and `-O` options at the same time. You can in this version of the compiler.

**Note –** Special Case: `-04 -g`. The combination `-04 -g` turns off inlining that you usually get with `-04`.

## *Link With Object Library: –llib*

The `–l` *lib* option links `ld`(1) with the object library *lib*.

## *Redirect the Object File: –o filename*

The `–o` option tells the compiler to name the generated executable *filename*. The default file name for executable files is `a.out`. The default file name for object files is the source file name with an `.o` extension. For example, the following command stores the executable in the file `myprog`.

```
hostname% pc -o myprog myprog.p
```

You may not give the object file the same name as the source file.

## *Run the Object Code Optimizer: –O[level]*

The `–O` option[1] tells the compiler to run the compiled program through the object code optimizer. The `–O` option also calls the `–P` option, which ensures boolean expressions are only evaluated as much as needed to determine the result. This causes an increase in compile time in exchange for a decrease in compiled code size and execution time.

There are four levels of optimization. You indicate the level by giving a digit from 1 to 4 after the `–O`. If you leave out the digit, the optimization level defaults to `–O2`.

The level numbers are interpreted as follows.

–O
: This is the most likely level of optimization to give fastest performance for most reasonable applications. Default = `–02`.

–O1, -xO1
: Do only the minimum amount of optimization (peephole). This is postpass assembly-level optimization. Do not use `–01` unless `-02` and `-03` result in excessive compilation time, or running out of swap space.

–O2, -xO2
Do basic local and global optimization. This is induction-variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, control-flow optimization, tail-recursion elimination, dead-code elimination, and tail-call elimination.

Level `-02` does not optimize references to or definitions of external or indirect variables. Level `-02` is the appropriate level for device drivers and programs that modify external variables from within signal handlers.

–O3, -xO3
Same as `-02`, but optimizes the uses and definitions of external variables. Level `-03` does not trace the effects of pointer assignments. Level `-03` should not be used when compiling device drivers or programs that modify external variables from within signal handlers.

–O4, -xO3
Same as `-03`, but traces the effects of pointer assignments and gathers alias information. Level `-04` should not be used when compiling device drivers or programs that modify external variables from within signal handlers.

---

**Note –** Levels `-03` and `-04` may result in an increase in the size of executables. When optimizing for size, use level `-02.` For most programs: `–04` is faster than `–03` is faster than `–02` is faster than `–01`. But in a few cases `–02` might be faster than the others, and `–03` might be faster than `–04`. You can try compiling with each level to see if you have one of these rare cases.

---

If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization and resumes subsequent procedures at the original level specified in the `–0` command-line option.

If you optimize at `–03` or `–04` with very large procedures (thousands of lines of code in a single procedure), the optimizer may require an unreasonable amount of memory. In such cases, performance of the machine may be degraded.

You can prevent this in the C-shell by limiting the amount of virtual memory available to a single process. To do this, use the limit command (see `csh` (1)). For example, to limit virtual memory to 16 megabytes:

```
tutorial%   limit datasize 16M
```

This causes the optimizer to try to recover if it reaches 16 megabytes of data space.

This limit cannot be greater than the machine's total available swap space, and in practice, should be small enough to permit normal use of the machine while a large compilation is in progress. For example, on a machine with 32 megabytes of swap space, the command `limit datasize 16M` ensures that a single compilation would never consume more than half of the machine's swap space.

The best setting of data size depends on the degree of optimization requested and the amount of real memory and virtual memory available. To find the actual swap space:

```
tutorial%   /usr/sbin/swap -s
```

To find the actual real memory:

```
tutorial%  /usr/sbin/prtconf | grep Memory
```

### *Implement Extended Language Features:* `-xl`

The `-xl` option implements a set of features that provide `broad` compatibility with Apollo Pascal. We recommend using `-xl` only when porting Pascal systems from Apollo platforms to SPARCsystem platforms. See the *Pascal 3.0.2 Reference Manual* for details of the features controlled by `-xl`.

Modules compiled with `-xl` are *not* compatible with modules compiled without `-xl`. You should not link these two types of modules together.

## *Options*

This section describes the `pc` command options. The options appear in alphabetical order.

## *Profile Execution:* `-a, -xa`

The `-a` option inserts code to count how many times each basic block is executed. It also calls a runtime recording mechanism that creates a `.d` file for every `.p` file (at normal termination). The `.d` file accumulates execution data for the corresponding source file. The `tcov`(1) utility can then be run on the source file to generate statistics about the program. For a complete description of the `tcov`(1) utility, see the man page.

## *Buffering of the* `Output` *File*: `-b`

It is inefficient for Pascal to send each character to a terminal as it generates its output. It is even less efficient if the output is the input of another program, such as the line printer daemon, `lpr`(1).

To gain efficiency, Pascal buffers output characters; it saves the characters in memory until the buffer is full and then outputs the entire buffer in one system interaction. By default, Pascal output is line buffered.

The `-b` option on the command line turns on block-buffering with a block size of 1024. You cannot turn off buffering from the command-line.

If you give the `-b` option in a comment in the program, you can turn off buffering or turn on block buffering. The valid values follow.

`{$b0}` No buffering.

`{$b1}` Line buffering.

`{$b2}` Block buffering. The block size is 1024.

Any number greater than 2 (for example, `{$b5}`) is treated as `{$b2}`. You can only use this option in the main program. The block buffering value in effect at the end of the main program is used for the entire program.

## *Shared Libraries:* `-B`*binding*

The `-B` option specifies whether libraries for linking are `static` (not shared, indicated with `-Bstatic`), or `dynamic` (shared, indicated with `-Bdynamic`).

Link editing is the set of operations necessary to build an executable program from one or more object files. *Static linking* indicates that the results of these operations are saved to a file. *Dynamic linking* refers to these operations when performed at runtime. The executable that results from dynamic linking appears in the running process, but is not saved to a file.

## *Suppress Linking:* `-c`

The `-c` option tells the compiler not to call the linker, `ld`(1). The Pascal compiler, `pc`, leaves a `.o`, or *object*, file for each source file. Without `-c`, `pc` calls the linker when it finishes compilation, and produces an *executable* file, which is called `a.out` by default.

## *Runtime Checks:* `-C`

The `-C` option enables the following runtime checks:

- Checks that subscripts and subranges are in range.
- Checks that the number of lines written to output does not exceed the number set by the `linelimit` procedure. (See the *Pascal 3.0.2 Reference Manual* for information on `linelimit`.)
- Checks for overflow, underflow, and divide-by-zero.
- Verifies the `assert` statement. (See the *Pascal 3.0.2 Reference Manual* for information on `assert`.)

If you do not specify `-C`, most runtime checks are disabled (divide-by-zero checks are always made) and the compiler treats the `assert` statement as a comment and never uses calls to the `linelimit` procedure to halt the program.

Note that the `-V0` and `-V1` options implicitly turn on `-C`.

## *Use C Data Alignment:* `-calign`

The `-calign` option tells the compiler to allocate storage in records the same way as the C compiler allocates structures. See the *Pascal 3.0.2 Reference Manual* for details of how data is aligned with and without `-calign`.

You can use `calign` within a program as well as on the command line. However, `calign` only has an effect in the `type` and `var` sections of the program. Any types you define when `calign` is on will use C-style alignment whenever you define variables of that type.

## *Generate Code for SPARCsystem Workstations:* `–cg89,-xcg89`

The `–cg89` floating-point option generates code to produce instruction scheduling and special instructions to exploit features in SPARC systems released in 1989 or later. Generates `fsqrts` and `fsqrtd` software root instructions directly.

## *Generate Code for SPARCsystem Workstations:* `–cg92,-xcg92`

The `–cg92` option generates code to produce instruction scheduling and special instructions to exploit features on SPARC systems released in 1992 or later. Use appropriate inline templates.

## *Conditional Compilation:* `-cond`

You can only use this option when you also use the `-xl` option.

The `–cond` option tells `pc` to compile the lines in your program that begin with the `%debug` compiler directive. If you compile your program without `–cond`, `pc` treats lines with the `%debug` directive as comments.

`–xl` runs your program through the preprocessor `cppas`, which handles the Apollo[1] DOMAIN-style Pascal compiler directives such as `%debug`.

See *Appendix A, "Pascal Preprocessor* for a complete description of conditional variables, `cppas`, and compiler directives.

## *Conditional Processing Variable:* `–config`

You can only use this option when you also use the `-xl` option.

1. Apollo, Aegis, and DOMAIN are registered trademarks of Apollo Computer Inc., a subsidiary of Hewlett Packard Company.

The –config option sets a conditional *variable* to true. You can only use this option when you use the preprocessor cppas, which is invoked when you use the -xl option.

Pascal supports the –config option with only one value. For example, Pascal accepts –config one, but not –config one two. To specify more than one variable, you may use multiple –config options on the command line.

When you use -config and do not give a variable, the value of the predefined conditional variable %config is set to *true.*

–xl runs your program through the preprocessor cppas, which handles the Apollo DOMAIN-style Pascal compiler directives such as %config.

See *Appendix A, "Pascal Preprocessor* for a complete description of conditional variables, cppas, and compiler directives.

## *Double Load/Store Instructions:* –dalign

The –dalign option tells the compiler to generate double load/store instructions wherever possible for faster execution. All double-typed data become double aligned, so this option should not be used when correct alignment is not assured.

## *Specify Static Binding:* -dn

Specify static linking in the link editor.

## *Show Compiler Passes:* -dryrun

The –dryrun option tells the compiler to show, but not execute, the commands constructed by the compilation driver. This allows you to see the order of execution of compiler passes without actually executing them.

## *Define Name to Preprocessor:* –D*name*[=*def*]

The –D option defines a symbol *name* to the C preprocessor, /usr/ccs/lib/cpp. It is equivalent to using the #define statement in your program. If you do not include a definition, *name* is defined as 1. See cpp(1) for more information.

If you use this option with the `-xl` option, `-D` is equivalent to using the `%config` directive in your program.

### *Specify Dynamic Binding:* `-dy`

Specify dynamic linking in the link editor.

### *Optimize for Speed:* `–fast`

The `–fast` option selects optimum compilation options for speed. Should provide close to the maximum performance for most realistic applications. A convenience option; it chooses the fastest code generation option available on the compile-time hardware, the optimization level `-02`, a set of inline expansion templates, the `-fnonstd` floating-point option. If you combine fast with other options, the last specification applies.

The code generation option, the optimization level and using inline template files can be overridden by subsequent switches. For example, although the optimization part of `-fast` is `-02`, the optimization part of `-fast -03` is `-03`. Do not use this option for programs that depend on IEEE standard exception handling; you can get different numerical results, premature program termination, or unexpected `SIGFPE` signals.

### *Non-Standard Floating Point:* `-fnonstd`

This option causes non-standard initialization of floating-point arithmetic hardware. By default, IEEE 754 floating-point arithmetic is nonstop, and underflows are gradual. (See the *Numerical Computation Guide* for a further explanation.) The `–fnonstd` option causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operations exceptions. These are converted into `SIGFPE` signals, and if the program has no `SIGFPE` handler, it will terminate with a memory dump.

`-fnonstd` also causes the math library to be linked in, by passing `-lm` to the linker.

### *Debugging Information:* `–g`

The `–g` option[1] tells `pc` to produce additional symbol table information for `dbx` and `debugger`.

### *Behavior Change: Debugging Optimized Code*

You can now compile using both the `-g` and `-O` options. However, there are some side effects:

- The `next` and `step` commands do not work, but the `cont` command does.
- If you have make files that rely on `-g` overriding `-O`, you will have to revise those files.
- If you have make files that check for a warning message that `-g` overrides `-O`, you must revise those make files.

---

**Note –** Special Case: `-O4 -g`. The combination `-O4 -g` turns off inlining that you usually get with `-O4`.

---

## *Display Help:* `–help`

The `–help` option lists and summarizes all available options.

## *Check Heap Pointers:* `–H`

The `–H` options tells `pc` to compile code to perform range checking on pointers into the heap. This option is implicitly turned on by the `–V0` and `–V1` options.

## *Include File Listing:* `–i` *name*

The `–i` option produces a listing for the specified procedure, function, `#include`, or `%include` file. For example, this command tells the compiler to make a listing of the routines in the file `scanner.i`.

```
hostname% pc –i scanner.i program.p
```

See `cpp`(1), or Chapter 5, “Program Construction and Management or Chapter 6, “Separate Compilation for information on include files.

---

1. In earlier versions of this Pascal, you could not use the `-g` and `-O` options at the same time. You can in this version of the compiler.

## *Specify Directories for Include Files: –*I*pathname*

The –I option gives the preprocessor additional places to look for #include and %include files. For example,

```
hostname% pc -I/home/incfiles -I/usr/incfiles program.p
```

The preprocessor searches for #include and %include files in this order:

- In /opt/SUNWspro/SC2.0.2/include
- In the directory containing the source file (except when you use the #include <*file*> form, in which case this directory is not searched)
- In directories named with –I options (if any), in left to right order
- In /usr/include

## *Make a Listing: –*l

The –l option enables a listing of the program as follows:

```
hostname% pc -l random.p
SPARCompiler Pascal PC -- Version SC3.0 08 Dec 1993 Pascal 3.0.2

Thu Feb 20 09:04 1993 random.p:
    1    program random_number(output);
    2    var
    4         i: integer;
    5         x: integer;
    6
    7    begin
    8         for i := 1 to 5 do begin
    9             write(trunc(random(x) * 101))
    10        end;
    11        writeln
    12   end.
```

The first line identifies the version of the compiler you are using. The next line gives the modification time of the file being compiled. The remainder of the listing is the source program.

### *Link With Object Library:* –l*lib*

The –l*lib* option links ld(1) with the object library *lib*.

### *In-line Templates:* –libmil, -xlibmil

The –libmil option tells the compiler to select the best inline templates for the floating-point option and operating system release available on this system.

### *Link with FORTRAN Startup Code:* –lpfc

The –lpfc option tells the compiler to link with common startup code for programs containing mixed Pascal and FORTRAN object files. Such programs should also be linked explicitly with the FORTRAN. See Chapter 8, "The FORTRAN–Pascal Interface for more information.

### *Map Identifiers and Keywords to lowercase:* –L

The –L option maps all keywords and identifiers to lowercase. In this Pascal, uppercase and lowercase are not interchangeable in identifiers and keywords. In standard Pascal, case is insignificant outside of character strings and character constants. The –L option is most useful for transporting programs from other systems. See also the –s option.

### *Pass* –R *Flag to Assembler:* –xMerge

The –xMerge option tells pc to call the assembler, as(1), with the –R option. This option merges the data segment of the resulting program with the text segment. See the *SunOS 5.2 Reference Manual* for more information on as(1).

### *Allow Misaligned Data:* –misalign

The –misalign option allows for misaligned data in memory. You should only use this option *if* you get a warning message that your data is misaligned.

---

**Note –** With the –misalign option, the compiler generates much slower code for references to formal parameters. If possible, recode the indicated section instead of recompiling your program with this option.

---

### *Best Available Floating Point:* –native

If you compile with the –native option, the compiler generates code for best floating-point hardware available on the machine you are compiling on.

### *Exclude all libraries:* -nolib, -xnolib

Does not link any libraries by default; that is, no -l options are passed to ld. Normally, the pc driver passes -lc to ld.

When you use -nolib, you have to pass all -l options yourself. For example:

```
pc -xnolib -Bstatic -lm -Bdynamic -lc test.p
```

links libm statically and the other libraries dynamically.

### *Exclude In-line Templates:* –nolibmil

The –nolibmil option tells the compiler to reset –fast so that it does *not* include inline templates. Use this *after* the –fast option.

```
pc –fast –nolibmil myprog.p
```

### *Do Not Queue License Requests:* -noqueue

The -noqueue option tells the compiler not to queue this compilation if a license is not available. Under normal circumstances, if no license is available, the compiler waits until one becomes available. With this option, the compiler returns immediately.

### *Disable Traceback:* -notrace

This option disables the runtime traceback. It is only effective when compiling the main program.

### *Redirect the Object File:* –o *filename*

The –o option tells the compiler to name the generated executable *filename*. The default file name for executable files is a.out. For example, the following command stores the executable in the file myprog.

```
hostname% pc -o myprog myprog.p
```

If you use this option with the `-c` option, the name you give is used as the name for the object file. The default file name for object files is the source file name with an `.o` extension. You may not give the object file the same name as the source file.

## *Run the Object Code Optimizer:* `–O[`*level*`]`

See the description of this option in Section , “Frequently Used Optionson page 27.

## *Generate an Execution Profile:* `–p` *and* `–pg, -xpg`

The `–p` and `–pg` options tell the compiler to produce code that counts the number of times each routine is called. The profiling is based on a periodic sample taken by the system, rather than by line counters.

### *Using the* `-p` *Option*

To generate an execution profile using the `–p` option:

1. Compile with the `–p` option.
2. Run `a.out` (which will produce a `mon.out` executable file).
3. Enter `prof a.out` The program prints a profile.

### *Using the* `-pg` *Option*

To generate an execution profile using the `–pg` option:

1. Compile with the `–pg` option.
2. Run `a.out` (which will produce a `gmon.out` executable file, a more sophisticated profiling tool than `mon.out`).
3. Enter `gprof a.out` The program prints a profile.

### *Partial Evaluation of Boolean Expressions:* –P

The –P option tells the compiler to use partial evaluation semantics on the boolean operators and and or. Left-to-right evaluation is guaranteed, and the second operand is evaluated only if necessary to determine the result.

### *Generate PIC Code:* –pic,-Kpic *and* –PIC,-KPIC

The -pic and -PIC options tell the compiler to generate position-independent code (PIC). One of these options should be used for objects which will be put into shared libraries. With PIC, each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table.

The size of the global offset table is limited to 8K with -pic. The -PIC option expands the global offset table to handle 32-bit addresses for those rare cases where there are too many data objects for -pic.

For more information on PIC see the section on shared libraries in the manual *SunOS 5.2 Linker and Libraries Manual.*

### *Pass Option:* –Qoption *Program Option*

The –Qoption passes *option* to *program*. The *option* value must be appropriate to that program and may begin with a plus or minus sign. The *program* value may be either fbe(1), (1cpp(1), cppas, inline(1), iropt, ld(1), pc0, or pc3. For example, the following command passes the option -R (allow recursive macros) to cpp:

```
hostname% pc -Qoption cpp -R myprog.p
```

### *Generate an Execution Profile:* -qp

This option is the same as -p option.

### *Use Alternate Path:* –Qpath *pathname*

The –Qpath option inserts *pathname* into the compilation search path. This provides an alternate path to search for each compiler component. You may select this option, for instance, to use a different linker or assembler. For

example, in the following command, pc searches /home/pascal/sparc for the compiler components and uses them if it finds them; if pc doesn't find the specified components, it uses the default components:

```
hostname% pc -Qpath /home/pascal/sparc testp.p
```

## *Produce Source Code:* –Qproduce

The –Qproduce option tells pc to produce source code of the type *sourcetype*, which can be one of the following:

.o Object file from as(1).

.pi Preprocessed Pascal source from cpp(1).

.s Assembler source. This is the same as the -S option.

For example, this command produces the file hello.s:

```
hostname% pc -Qproduce .s hello.p
```

## *Specify Library Search Path:* -R *path [ :dir ]*

This option passes a colon-separated list of directories that specify the library search path used by the run-time linker. If present and not null, it is recorded in the output object file and passed to the run-time linker.

If both LD_RUN_PATH and the -R option are specified, the -R option takes precedence.

## *Standard Pascal Only:* –s[*level*]

The –s option tells the compiler to accept standard Pascal only. This Pascal has two levels of compliance with standard Pascal: Level 0 and Level 1. Level 1 is the same as Level 0 but also allows conformant arrays.

You can specify the level of compliance as follows:

–s0 Accept Level 0 compliance with standard Pascal.

–s1 Accept Level 1 compliance with standard Pascal.

–s Same as –s1.

This option causes many features of this Pascal that are not found in standard Pascal to be diagnosed with warning messages. These features include nonstandard procedures and functions, extensions to the procedure `write`, the padding of constant strings with blanks, and preprocessor directives.

In addition, all letters, except character strings and constants, are mapped to lowercase. Thus, the case of keywords and identifiers is effectively ignored.

This option is most useful when a program is to be ported to other machines.

### *SourceBrowser Information:* `–sb, -xsb`

The `–sb` option produces additional table information for the SourceBrowser.

### *SourceBrowser Information Without Compiling:* `–sbfast, -xsbfast`

The `–sbfast` option does the same as `-sb` but does not compile.

### *Generate Assembly Language:* `–S`

The `–S` option compiles the program and outputs the assembly language in the file *sourcefile*`.s`. For example, the following command places the assembly language translation of `rmc.p` in the file `rmc.s`. No executable file is created.

```
hostname% pc –S rmc.p
```

### *Generate* `pc3` *Stab Information:* `-tc`

The `-tc` option tells the compiler to generate `pc3` stab information to allow cross-module type checking.

### *Directory for Temporary Files:* `–temp=`*dir*

The `–temp` option tells `pc` to locate the temporary files that it creates during compilation in the directory named *dir*. For example, the following command puts the temporary files in the current directory.

```
hostname% pc -temp=. hello.p
```

If you do not specify a temporary directory, `pc` places the temporary files in the `/tmp` directory.

## *Report Execution:* `-time`

The `-time` option tells the compiler to report execution performance statistics for the various compilation passes. Here is some sample output. (Some spaces have been removed so the output would fit on the page.)

```
hostname% pc -time hello.p
cpp:time U:0.0s+S:0.1s=0.2s REAL:1.6s 11%. core T:0k D:0k. io IN:4b OUT:3b. pf IN:25p OUt:184p.
pc0:time U:0.0s+S:0.3s=0.4s REAL:3.2s 13%. core T:0k D:4k. io IN:4b OUT:4b. pf IN:70pOUT:131p.
cg: time U:0.0s+S:0.1s=0.2s REAL:2.0s 12%. core T:0k D:1k. io IN:2b OUT:1b. pf IN:39p OUT:163p.
as: time U:0.0s+S:0.2s=0.3s REAL:1.5s 19%. core T:0k D:1k. io IN:3b OUT:10b.pf IN:33pOUT:117p.
pc3:time U:0.1s+S:0.1s=0.3s REAL:0.9s 31%. core T:0k D:1k. io IN:7b OUT:0b. pf IN:20pOUT:109p.
ld:time U:0.8s+S:0.9s=1.8sREAL:10.2s 17%. core T:0k D:21k.io IN:74bOUT:29b.pf IN:89pOUT:184p.
```

Each line begins with the name of the compiler pass. The rest of the line is divided into four parts: `time`, `core`, `io`, and `pf`.

Time gives the time used by that pass of the compiler, in this order.

1. User time
2. System time
3. Total CPU time, which is the sum of user and system time
4. Real (clock) time
5. Percent of real time used by CPU time

Core gives memory usage statistics for the pass, in this order:

1. The first item is always 0, and currently has no meaning.
2. The second item is the integral resident set size.

The `io` section gives the volume of input and output operations, expressed in blocks.

The `pf` section gives the amount of page faults, expressed in pages, in this order:

1. Page faults not requiring physical I/O
2. Page faults requiring physical I/O

### *Undefine Name:* –U *name*

The –U option removes any initial definition of the cpp(1) symbol *name*. See cpp(1) for more information. You cannot use this option with the -xl option.

### *Print Names:* –v

The –v (verbose) option prints the command line used to call each compilation pass.

### *Print Versions:* –V

The –V option prints the version number of each compilation pass.

### *Verification:* –V0 *and* –V1

The –V0 and –V1 options are turn on sets of options that insert checks into the object file and as follows:

–V0 Equivalent to -C, -H, -L, and -s0.

–V1 Equivalent to –C, –H, –L, and –s1.

### *Suppress Warning Diagnostics:* –w

By default, the compiler prints warnings about inconsistencies it finds in the input program. The –w option turns off the warnings.

To turn off warnings in a program comment, use this form:

```
{$w-}
```

### *Function-Level Reordering:* –xF

This option is only available in the version of Pascal for the SunOS 5.x operating system.

The –xF option enables performance analysis of the executable file using the SPARCworks Analyzer and Debugger. This option also causes the assembler to generate some debugging information in the object file, necessary for data collection. The compiler generates code that can be reordered at the function

level. This takes each function in the file and places it into a separate section. For example, functions `fcn1()` and `fcn2()` are placed in the sections `.text%fcn1` and `.text%fcn2`. You can control the order of functions in the final executable by using `–xF` and the loader `–Mmapfile` option.

In the map file, if you include the flag O in the string of segment flags, then the static linker ld will attempt to place sections in the order they appear in the mapfile. See the *SunOS 5.2 Linker and Libraries Manual* for more about this option, the segment flags, and the map file.

## *Implement Extended Language Features:* `–xl`

The `–xl` option implements a set of features that provide `broad` compatibility with Apollo Pascal. We recommend using `–xl` only when porting Pascal systems from Apollo platforms to SPARCsystem platforms. See the *Pascal 3.0.2 Reference Manual* for details of the features controlled by `-xl`.

When you use `-xl`, the compiler invokes the `cppas` preprocessor in place of `cpp`(1). See Appendix A, “Pascal Preprocessor for information on `cppas`.

Modules compiled with `–xl` are *not* compatible with modules compiled without `–xl`. You should not link these two types of modules together.

## *Get License Information:* `–xlicinfo`

The `-xlicinfo` option returns information about the licensing system. In particular, it returns the name of the license server and the user ids of users who have licenses checked out. When you give this option, the compiler is not invoked and a license is not checked out.

## *Implement Symbol Tables for* `dbx: –xs`

The `-xs` option disables *autoload* for `dbx`. This is in case you cannot keep the `.o` files around. This passes the `-s` option to the assembler and the linker.

*No Autoload*: This is the older way of loading symbol tables.

- The compiler instructs the linker to place all symbol tables for `dbx` in the executable file.
- The linker links more slowly and `dbx` initializes more slowly.

- If you move the executables to another directory, then to use dbx you must move the source files, but you do not need to move the object (.o) files.

*Autoload:* This is the newer (and default) way of loading symbol tables.

- Distribute this information in the .o files so that dbx loads the symbol table information only if and when it is needed.
- The linker links faster and dbx initializes faster.
- If you move the executables to another directory, then to use dbx you must move both the source files and the object (.o) files.

## *Initialize Local Variables to Zero: –Z*

The –Z option tells pc to insert code that initializes all local variables to zero. (Standard Pascal does not allow initialization of variables.)

# The File System 4

This chapter is a basic introduction to the file system and how it relates to the Pascal I/O system. Topics covered include:

- Hierarchy
- Directories
- File names and path names
- Standard input and standard output
- Redirection
- Piping

If you understand these concepts, then skip this chapter. For a more detailed discussion of the file system structure, see the *SunOS User's Guide: Getting Started.*

## Hierarchy

The file system has an inverted tree structure. The top of the file system is the root directory. Directories, subdirectories, and files all branch *down* from the root. Directories are nodes on the tree, and any directory can have subdirectories or ordinary files branching down from them. The only directory that is not a subdirectory is the root directory, so a distinction is not usually made between directories and subdirectories.

Figure 4-1 shows a diagram of the file system tree structure. This figure is greatly simplified. On any actual system, the root directory would contain many, many more directories.

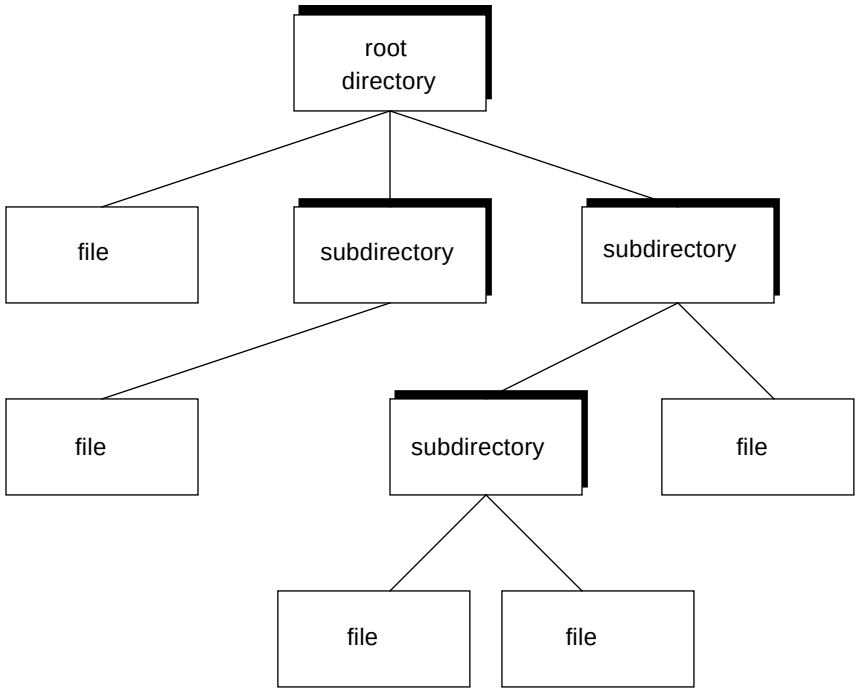


*Figure 4-1* File System Hierarchy

## *Directories*

Directories are really just files that can contain other files. Directory names are formed just like filenames. The rules for filenames are described in the next section, “Filenames and Pathnames.

When you are logged in, you are always “in” some particular directory, which means you are always in some particular part of the hierarchy.
The directory you are in is called your *working directory*, or sometimes the current working directory or current directory. The period (.) symbol refers to the current working directory.

Every directory, except the root directory, is in a directory, often called the *parent* directory. The symbol `..` (two periods) refers to the parent of the current directory.

When you first log on, you are usually in your *home* directory. If you are using the C shell, the symbol ~ (tilde) refers to your home directory.

You can move to a different directory using the `cd` (change directory) command. You can change to a directory using a pathname (described in the next section, “Filenames and Pathnames). You can also change directories using the symbols `..` and ~. For example, this command moves to the parent of the current working directory.

```
hostname% cd ..
```

This command moves to your home directory.

```
hostname% cd ~
```

You can also move to your home directory by executing `cd` with nothing after it.

You can find out what directory you are in using the `pwd` (print working directory) command. For example,

```
hostname% pwd
/home/somesystem/yourname
```

You can get the name of the working directory or change the working directory from within programs by calling C library routines from your program. (Pascal has no routines for these operations.)

Additional explanations of the file system organization and relevant shell commands are located in the *SunOS User’s Guide: Getting Started*.

# 4

## *Filenames and Pathnames*

Every file has a *filename*. A filename can be up to 512 characters long. Names can be made of most characters, but you should restrict your use of punctuation in filenames to the dot (.), underscore ( _ ), comma (,), plus (+), and minus (-).

A *pathname* prepends the location of the file to the filename.

Since directories are also files, they also have names that are formed in the same way. The single exception is the name of the root directory. The root directory is called / (pronounced *slash*).

To understand how pathnames are constructed, consider the sample hierarchy shown in Figure 4-2.

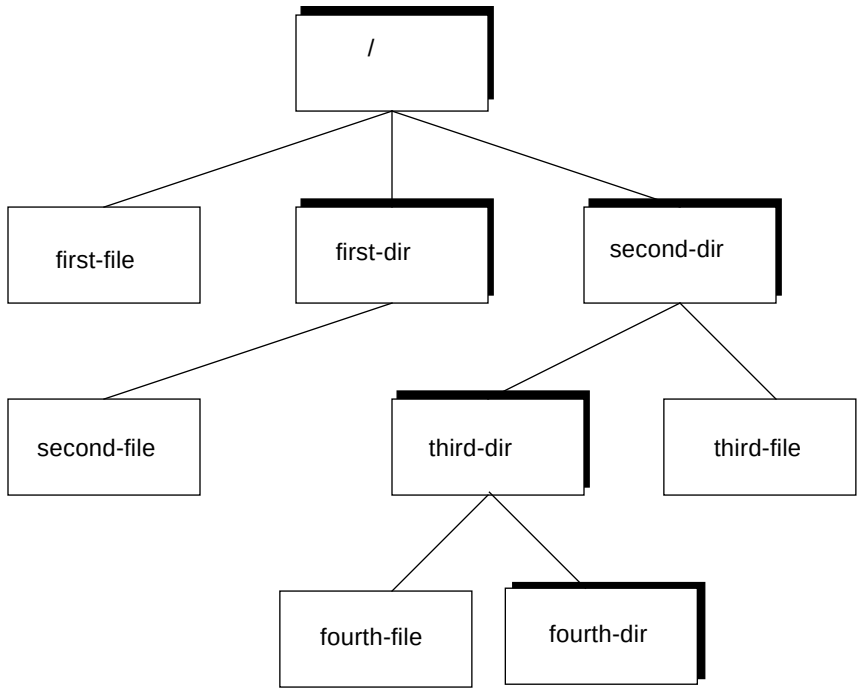


*Figure 4-2* A File Path

There are *absolute* pathnames and *relative* pathnames.

An absolute pathname describes the location of the file by naming all the directories in the path from the file to the root. For example, the absolute path for `first-file` is `/first-file`. For `fourth-file`, the absolute pathname is `/second-dir/third-dir/fourth-file`.

A relative pathname describes the location of a file in relation to your current position in the hierarchy. For example, if your current working directory is `second-dir`, the relative pathname for third-file is simply `third-file` and the relative pathname for fourth-file is `third-file/fourth-file`.

Notice that the slash (/) character has two uses: as the name of the root directory and as the separator for the components of a pathname. All absolute pathnames start with /. All pathnames that do not start with / are relative path names.

A pathname can have any number of components, but can only be up to 1024 characters long.

You can change directories using either absolute or relative pathnames. For example, if you are in the directory `second-dir`, this command uses an absolute pathname to change to directory `fourth-dir`.

```
hostname% cd /second-dir/third-dir/fourth-dir
```

This command changes to the same directory using a relative pathname.

```
hostname% cd third-dir/fourth-dir
```

## *Standard Input and Standard Output*

The Pascal I/O standard input and standard output files, `input` and `output`, are mapped to the input and output stream of the shell where the program starts. Consider the following program.

```
program std (input, output);
var x:integer;
begin
    write ('Type a number. ');
    read (x);
    writeln (x)
end.
```

When you compile this program and run it in a shell window, it produces output as shown below.

```
hostname% a.out
Type a number. 322
       322
hostname%
```

## *Redirection*

You can redirect standard input and standard output so that your program gets input from a file or sends output to a file. Redirection is a way of changing the files that a program uses without passing a filename to the program.

The symbol for redirecting standard input is the "<" sign, and for standard output is the ">" sign.

File redirection is a function performed by the command interpreter or shell when a program is invoked by it. Both standard input and standard output may be redirected to and from different files on the same command line.

Standard error can also be redirected so it does not appear on your workstation display. In general, this is not a good idea, since you usually want to see error messages from the program immediately, rather than sending them to a file. The shell syntax to redirect standard error varies, depending on whether you are using the Bourne shell or the C shell. Refer to the *SunOS User's Guide: Getting Started* for more information on redirecting standard error.

### *Input*

The shell command line:

```
hostname% myprog < mydata
```

causes the file `mydata` (which must already exist) to be connected to the standard input of the program `myprog`.

### *Output*

Similarly, the shell command line:

```
hostname% myprog > myoutput
```

causes the file `myoutput` (which is created if it does not exist, or overwritten if it does) to be connected to the standard output of the program `myprog` when it is run. So if the Pascal program `myprog` writes to unit 6, it writes to the file `myoutput`.

### *Output/Append*

Similarly, the shell command line:

```
hostname% myprog >> myoutput
```

causes the output of `myprog` to be appended to the file `mypoutput` (which is created if it does not exist).

## *Piping*

You can connect the standard output of one program directly to the standard input of another without using an intervening temporary file. The mechanism to accomplish this is called a *pipe*. A shell command line using a pipe looks like this:

```
hostname% firstprog | secondprog
```

This causes the standard output of `firstprog` to be piped to the standard input of `secondprog`. Piping and file redirection can be combined in the same command line. A simple example is as follows:

```
hostname% myprog < mydata | wc > datacount
```

In the above program, `myprog` takes its standard input from the file `mydata`, and has its standard output piped into the standard input of the `wc` command, the standard output of which is redirected into the file `datacount`.

# *Program Construction and Management* 5

This chapter is an introduction to the methods generally used to construct and manage programs using SPARCompiler Pascal.

## *Units*

For many reasons, it is often inconvenient to store a program in a single file. For example, some programs are too large to be conveniently handled in a single file.

You can break a program up in a number of ways. Perhaps the simplest is to use an *include file*. An include file is a separate file that is copied in by the compiler when it encounters an `include` compiler directive. For example, consider the following program:

```
program include (output);
#include "includefile"
```

The line `#include "includefile"` is a *compiler directive* to `cpp`(1), which is the Pascal compiler's preprocessor. The directive tells `cpp`(1) to find the file `includefile` and copy it into the stream before continuing.

The actual `includefile` looks like this:

```
begin
      writeln ('Hello, world.')
end.
```

In this example, the include file contains the entire program. In reality, an include file would probably contain a set of variable or procedure declarations. Include files are often used when a set of declarations needs to be shared among a number of programs.

However, suppose your program was very large, and took a long time to compile. Using include files might make editing more convenient, but when you made a change in one part of your program you would still need to recompile the entire program. As another example, suppose you wanted to be able to share compiled code with other people, but for reasons of security or convenience, did not want to share the source code.

Both of these problems are solved by *separately compiled units*, generally called units. A unit is a part of a program stored in its own file and linked with the rest of the program after compilation.

## *Program Units and Module Units*

There are two kinds of units: *program units* and *module units*.

A program unit looks like any program. It begins with a program header and contains the main program.

Here is an example of a program unit:

```
program program_unit (output);
procedure say_hello; extern;
begin
      say_hello
end.
```

Notice that the body of the procedure `say_hello` is not defined in this program unit, but the program unit does contain a declaration of the interface

to the procedure. The keyword `extern` declares that `say_hello` is declared in a module unit.[1]

A module unit can begin with an optional module header, followed by a set of compilable declarations and definitions. Modules that call externally defined routines need to have declarations for those routines.

```
module module_unit;
procedure say_hello;
begin
      writeln ('Hello, world.')
end;
```

Every program must have one and only one program unit; a program can have any number of module units. Any unit can call procedures declared in any other unit; each unit must have `external` declarations for every procedure it uses that is not defined in that unit.

A module unit can also be used as a *library*, that is, as a collection of useful routines that is shared among a number of programs.

## *How to Compile When You Use Units*

Consider the units given in the previous section, “Program Units and Module Units. You can compile and link these units on a single line.

```
hostname% pc program_unit.p module_unit.p
```

This command produces the executable object `a.out`.

1. A statement that shows the interface of a routine is called a *declaration*, because it declares the name and parameters of the routine. The set of statements that shows the entire routine, including the body, is called the *definition* of the routine. There must be only one definition for a given routine, but every routine must be declared in every module or program unit that uses it.

You can also separate the compilation and linking/loading steps.

```
hostname% pc program_unit.p -c
hostname% pc module_unit.p -c
hostname% pc program_unit.o module_unit.o
```

In this case, you call `pc` on each unit with the "compile only" option (`-c`), which produces an object file with the extension `.o`. When you use this option, the compiler driver does not call the linker/loader `ld`. You then call `pc` a second time, giving the names of the object files, and `pc` calls `pc3` to check for name and type conflicts and then calls the linker/loader.

Note that calling the linker/loader `ld`(1) directly does not have the same effect as calling `pc`; when you call `ld`(1) directly, the files are linked and loaded, but they are not checked for conflicts.

## *Units and Header Files*

A complex program may have many routines defined in modules. Each routine must have a declaration (for example, `procedure proc; extern;`) in each file that calls the routine. The easiest way to be sure that you have a correct and consistent set of declarations is to create a *header file*.

A header file is a file that contains a set of declarations, and nothing else. You use a header file by using an `include` directive to include the header file in the compilation.

For example, here is the program `program_unit` modified to use a header file:

```
program program_unit2 (output);
include "header.h"
begin
      say_hello
end.
```

In this case, the content of `header.h` is very simple:

```
procedure say_hello; extern;
```

In a real program, `header.h` would probably have many declarations in it and would be included in several modules. Aside from routine declarations, header files often contain constant, type, and variable declarations.

## *Sharing Variables Between Units*

Variables global across a unit (that is, not declared locally in a routine) can be public or private. A public variable can be shared by any unit that is linked to the unit that declares the variable. A private variable cannot be shared.

You can use the `public` and `private` reserved words to declare that a `var` section declares public or private variables. For example:

```
program program_unit3 (output);
public var
      x : integer;
private var
      y : integer;
```

When you don't use `public` or `private`, variables are public by default. However, when you compile with the `-xl` option, variables are private by default.

To share a public variable, simply declare it in each unit where you want to share it. As long as the variable is public, each reference to that variable will access the same data.

Here is a program unit that declares a variable.

```
program program_unit3 (output);
var
      x : integer;

procedure say_hello; external;

begin
      for x := 1 to 5 do say_hello
end.
```

Here is a module unit that declares a variable with the same name.

```
module module_unit3;
var
      x : integer;

procedure say_hello;

begin
      writeln ('Hello, world for the', x, ' time.')
end;
```

By default, both definitions of variable `x` are public. Thus, when you compile and link the program and module units, references to `x` refer to the same variable.

```
hostname% pc program_unit3.p module_unit3.p
program_unit.p:
module_unit.p:
Linking:
hostname% a.out
Hello, world for the 1 time.
Hello, world for the 2 time.
Hello, world for the 3 time.
Hello, world for the 4 time.
Hello, world for the 5 time.
```

If you compile the program giving the `-xl` option, the variables are private by default.

```
hostname% pc -xl program_unit.p module_unit.p
program_unit.p:
module_unit.p:
Linking:
hostname% a.out
Hello, world for the 0 time.
Hello, world for the 0 time.
Hello, world for the 0 time.
Hello, world for the 0 time.
Hello, world for the 0 time.
```

You can get the same effect by explicitly declaring the variable in a `private var` section. Similarly, when you use `-xl`, you can create public variables by declaring them in a `public var` section.

As with routine declarations, it is often a good idea to declare public variables in an *include* file. Doing so makes it easier to keep your declarations consistent.

There are other methods for making variables visible to different units. See Chapter 6, “Separate Compilation for more information.

## *Libraries*

As mentioned before, you can use a module unit as a library of useful functions. The simplest way to do so is to create a source file containing the definitions of your library routines and then compile it using the `-c` option. You can then link the resulting `.o` file to any number of files. For convenience, you probably should create a header file containing the routine declarations for the library.

### *Different Kinds of Libraries*

The kind of simple library described above has two distinct problems.

- When a library gets large, it may be inconvenient to deal with the library’s source in a single file, both for ease of editing and so you can avoid recompiling a large file when you only change part of it.

  On the other hand, it would be inconvenient to have to name many library modules on the command line when you link your program. Thus, it would be helpful to have a way of combining a number of library modules.

- Several programs that a user might run at the same time might share the same library. Under the simple scheme described above, each program would have its own copy of the library. It would save space and might save I/O time if several programs could share library code.

Both problems have solutions. First, you can combine, or *archive* modules together. Second, you can create a *shared library.*

See *SunOS 5.2 Linker and Libraries Manual* for information on creating archived and shared libraries.

# 5

# *Separate Compilation* 6

This chapter describes separate compilation in Pascal. Chapter 5, “Program Construction and Management gives an introduction to the concepts in this chapter.

In separate compilation, a program is divided into several units that may be separately compiled into object (`.o`) files. The object files are then linked using `pc`, which invokes `pc3` to check for the consistent use of global names and declarations across the different units and then invokes `ld`(1) to link and load the units. (You can also give `pc` the names of all the units at once, in which case `pc` compiles all the units, checks for consistency, and links the units in one step.)

Separate compilation differs from independent compilation in that, in independent compilation, you invoke `ld` directly, so that there is no consistency checking. Independent compilation is not addressed in the Pascal documentation.

## *Units*

This version of Pascal provides two types of source files or units: the program unit and the module unit.

### *Program Unit*

The program unit is the source program with the program header. It consists of the following components:

```
<program unit> ::= <program heading> <declaration list> <program body>
```

Each program you write may have only one program unit. The *program body* is the first code that Pascal executes.

### *Module Unit*

A module unit is a source program that does not have a program header. A module unit has the following syntax:

```
<module unit> ::= [ <module heading> ] <declaration list>
```

The module heading contains the reserved word `module` followed by an identifier:

```
<module heading> ::= [ 'module' <identifier> ';' ]
```

For example

```
module sum;
```

is a legal module heading. The module heading is optional.

## *Sharing Variables and Routines across Multiple Units*

This Pascal supports three methods of sharing variables and routines between units:

- Include files
- Multiple variable declarations
- `extern/define` variable declarations.

These methods are not mutually exclusive; for example, you can declare a variable as either `extern` or `define` in an `include` file.

All three methods are described in the following sections. For information on the differences in separate compilation in earlier versions of Pascal, see the *Pascal 3.0.2 Reference Manual*.

## *Compiling withou the* `-xl` *Option*

This section describes three ways of sharing variables and routines across units when you compile your program without the `-xl` option.

### *Sharing Public Variables*

If you declare a variable in two or more separate units and the variable is public in both places, that variable is shared between units. Variables are public by default, unless you compile with the `-xl` option, in which case variables are private by default. In this example, the variable `global` is public by default, and thus shared between the program and the module.

The program unit, `shrvar_prog.p`:

```
program shrvar_prog;

var
    global: integer;

procedure proc; external;

begin { program body }
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global)
end. { shrvar_prog }
```

The module unit, `shrvar_mod.p`: The assignment of a new value to `global` and `max_array` in the procedure `proc` in `shrvar_prog.p` is repeated in `shrvar_mod.p`.

```
module shrvar_mod;

var
   global: integer;

procedure proc;

begin
    writeln('From PROC: ',global);
    global := global + 1
end; { proc }
```

The commands to compile and execute `shrvar_prog.p` and `shrvar_mod.p`.

```
hostname% pc shrvar_prog.p shrvar_mod.p
shrvar_prog.p:
shrvar_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC: 1
From PROC             : 1
From MAIN, after PROC : 2
hostname%
```

## *Using* `extern` *Option to Share Routines*

If a program or module calls a procedure not defined in that unit, you must declare it with either the `extern` or `external` routine option. For instance, in the previous example, the procedure `proc` is defined in `shrvar_mod.p`, but used in `shrvar_prog.p`. Thus, it is declared as `external` in `shrvar_prog.p`. Also, `proc` must also be defined as public in `shrvar_mod.p`, which happens by default.

## *Using* `include` *Files to Share Variables and Routines*

The `include` file contains the declarations for the program. Placing all program declarations in a single file makes your program more consistent and easier to maintain.

To use this feature, place the number sign character (#) in the first position of a line immediately followed by the word `include`, and then a filename enclosed in angle brackets (< and >) or double quotation marks("). The different enclosures (<> and "") affect the search order for files. The syntax for the `#include` directive is determined by `cpp`(1).

When the compiler encounters the `#include` in the input, it inserts the lines from the included file into the input stream.

The program unit, `inc_prog.p`, which includes the file `include.h`.

```
program inc_prog;

#include "include.h"

begin { program body}
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global)
end. { inc_prog }
```

The module unit, `inc_mod.p`, which also includes the file `include.h`.

```
module inc_mod;

#include "include.h"

procedure proc;

begin
    writeln('From PROC            : ', global);
    global := global + 1
end; { proc }
```

The `include` file, `include.h`:

```
var
    global : integer;

procedure proc; extern;
```

The commands to compile and execute `inc_prog.p` and `inc_mod.p`:

```
hostname% pc inc_prog.p inc_mod.p
inc_prog.p:
inc_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC:     1
From PROC :                 1
From MAIN, after PROC:      2
hostname%
```

## *Using the* `-xl` *Option*

When you use the `-xl` option, variables and top-level procedures and functions declared in the program unit default to private. Look at the difference when you compile and execute `shrvar_prog.p` and `shrvar_mod.p` with `-xl`. (The code is listed in the section "Sharing Public Variables.)

The commands to compile and execute `shrvar_prog.p` and `shrvar_mod.p` with the `-xl` option.

```
hostname% pc -xl shrvar_prog.p shrvar_mod.p
shrvar_prog.p:
shrvar_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC: 1
From PROC : 0
From MAIN, after PROC: 1
hostname%
```

Without `-xl`, the variable `global` in `shrvar_mod.p` was treated as public; here, `global` is treated as private. Thus, the assignment

```
global := global + 1;
```

is not reflected in `shrvar_prog.p`; instead, each file uses its own private copy of `global`.

The following sections describe five ways of sharing variables and routines across units when you compile your program with `-xl`.

## *Using* `public var` *Declarations*

The following examples uses the `public` attribute in the var declaration to make `global` public when you compile your program with `–xl`.

The program unit, `pubvar_prog.p`, which declares `global` as `public`.

```
program pubvar_prog;

public var
    global: integer;

procedure proc;
    external;

begin
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global)
end. { pubvar_prog }
```

The module unit, `pubvar_mod.p`, which also declares `global` as `public`.

```
module pubvar_mod;

public var
    global : integer;

procedure proc;

begin
    writeln('From PROC             :',global);
    global := global + 1;
end; { proc }
```

The commands to compile and execute `pubvar_prog.p` and `pubvar_mod.p`:

```
hostname% pc -xl pubvar_prog.p pubvar_mod.p
pubvar_prog.p:
pubvar_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC: 1
From PROC : 1
From MAIN, after PROC: 2
hostname%
```

### *Using the* `define` *Variable Attribute*

This example makes global public using the define attribute of the variable declaration.

The program unit, `defvar_prog.p`:

```
program defvar_prog;

var
    global: extern integer;

procedure proc;
    external;

begin
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global);
end. { defvar_prog }
```

The module unit, `defvar_mod.p`, which makes `global` public using the `define` attribute.

```
module defvar_mod;

var
    global : define integer;

procedure proc;

begin
    writeln('From PROC            : ',global);
    global := global + 1;
end; { proc }
```

The commands to compile and execute `defvar_prog.p` and `defvar_mod.p`:

```
hostname% pc -xl defvar_prog.p defvar_mod.p
defvar_prog.p:
defvar_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC:     1
From PROC             :     1
From MAIN, after PROC :     2
```

## *Using the* `define` *Declaration*

This example defines `global` in the module `defvar_mod2` using the `define` declaration. The advantage of using the `define` declaration over the `define` variable attribute is that the `define` declaration can be easily converted to use include files.

The program unit, `defvar_prog.p`:

```
program defvar_prog;

var
    global: extern integer;

procedure proc;
    external;

begin
    global := 1;
    writeln('From MAIN, before PROC: ', global);
    proc;
    writeln('From MAIN,  after PROC: ', global)
end. { defvar_prog }
```

The module unit, `defvar_mod2.p`, which defines `global` in a `define` declaration.

```
module defvar_mod2;

var
    global : extern integer;

define
    global;

procedure proc;

begin
    writeln('From PROC            : ',global);
    global := global + 1;
end; { proc }
```

The commands to compile and execute `defvar_prog.p` and `defvar_mod2.p`:

```
hostname% pc -xl defvar_prog.p defvar_mod2.p
defvar_prog.p:
defvar_mod2.p:
Linking:
hostname% a.out
From MAIN, before PROC:     1
From PROC             :     1
From MAIN, after PROC :     2
```

## *Using* include *Files*

In the following example, the `extern` declaration for the variable `global` is in the `include` file `inc_prog2.p` and is therefore included in both files. The `define` declaration in file `inc_mod2.p` cancels the `extern` definition.

The program unit, `inc_prog2.p`:

```
program inc_prog2;

%include "include2.h";

procedure proc; extern;

begin
    global := 1;
    writeln('From MAIN, before PROC: ',global);
    proc;
    writeln('From MAIN,  after PROC: ',global);
end. { proc }
```

The module unit, `inc_mod2.p`:

```
module inc_mod2;

define
    global;

%include "include2.h";

procedure proc;

begin
    writeln('From PROC             : ',global);
    global := global + 1;
end; { proc }
```

The `include` file, `include2.h`:

```
var
    global : extern integer;
```

The commands to compile and execute `inc_prog2.p` and `inc_mod2.p`:

```
hostname% pc -xl inc_prog2.p inc_mod2.p
inc_prog2.p:
inc_mod2.p:
Linking:
hostname% a.out
From MAIN, before PROC:  1
From PROC             :  1
From MAIN, after PROC :  2
hostname%
```

## *Using* `extern`

In the previous example, the `extern` definition for variables was put into an `include` file and then shared. You can do the same for the `extern` procedure definition. When you do this, you must also declare the variable with a `define` declaration in the module that defines the procedure. This nullifies the effect of the `extern` declaration.

The program unit, `ext_prog.p`:

```
program ext_prog;

%include "extern.h";

begin
    global := 1;
    writeln('From MAIN, before PROC: ',global);
    proc;
    writeln('From MAIN,  after PROC: ',global);
end. { ext_prog }
```

The module unit, `ext_mod.p`:

```
module ext_mod;

define
    global, proc;

%include "extern.h";

procedure proc;

begin
    writeln('From PROC : ',global);
    global := global + 1;
end; { proc }
```

The `include` file, `extern.h`:

```
var
    global : extern integer;

procedure proc; extern;
```

The commands to compile and execute `ext_prog.p` and `ext_mod.p`:

```
hostname% pc -xl ext_prog.p ext_mod.p
ext_prog.p:
ext_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC: 1
From PROC : 1
From MAIN, after PROC: 2
hostname%
```

## *Sharing Declarations in Multiple Units*

The `extern` and `external` directives for procedure and function declarations allow you to optionally specify the source language of a separately compiled procedure or function. For example, `extern fortran` directs the compiler to

generate calling sequences compatible with SPARCompiler FORTRAN. Then, `external c` directs the compiler to generate calling sequences compatible with SPARCompiler C.

For routines declared `extern fortran` or `external fortran`, the changes in the calling sequence are as follows:

- For value parameters, the compiler creates a copy of the actual argument value in the caller's environment, and passes a pointer to the temporary variable on the stack. Thus, you don't need to create (otherwise useless) temporary variables.
- The compiler appends an underscore to the name of the external procedure to conform to a naming convention of the `f77`(1) compiler. Names of Pascal procedures called from FORTRAN must supply their own trailing underscore (_).
- Multidimensional Pascal arrays are not compatible with FORTRAN arrays. Because FORTRAN uses column-major ordering, a multidimensional Pascal array passed to FORTRAN appears transposed.

For routines declared `extern c` or `external c`, the changes in the calling sequence are as follows:

- Value parameters of type `shortreal` are treated as `double` or `longreal`.
- A warning is generated if you attempt to pass a nested function.

When you compile your program with the `–xl` option, you can also use `nonpascal` to declare non-Pascal routines when porting DOMAIN programs written in DOMAIN Pascal, FORTRAN, and C.

# *The C–Pascal Interface* 7

This chapter describes how to mix C and Pascal modules in the same program.

## *Compiling Mixed-Language Programs*

You must use the compiler option `–lpc` when you compile a C main routine that calls Pascal. `–lpc` includes the Pascal object library `libpc`. For example:

```
hostname% pc -c -calign my_pascal.p
hostname% cc my_pascal.o my_c.c -lpc
hostname%
```

The `-c` option produces an unlinked object file. The `-calign` option causes `pc` to use C-like data formats for aggregate objects.

When you compile a Pascal main routine that calls C, you don't have to use any special options, but the `-calign` option is again useful. The C object library, `libc`, is automatically brought in for every Pascal compilation. You can check this by running `pc` with the `–v` option.

For example:

```
hostname% cc -c my_c.c
hostname% pc -calign my_c.o my_pascal.p
hostname%
```

# 7

## *Compatibility of Types for C and Pascal*

Table 7-1 and Table 7-2 list the default sizes and alignments of compatible types for C and Pascal.

*Table 7-1* C / Pascal Size and Alignment of Compatible Types

| Pascal Type | C Type | Size (bytes) | Alignment (bytes) |
|---|---|---|---|
| double | double | 8 | 8 |
| longreal | double | 8 | 8 |
| real | double | 8 | 4 |
| single | float | 4 | 4 |
| shortreal | float | 4 | 4 |
| integer16 | short int | 2 | 2 |
| integer32 | int | 4 | 4 |
| integer | int | 4 | 4 |
| -128..127 | char | 1 | 1 |
| boolean | char | 1 | 1 |
| alfa | char a[10] | 10 | 1 |
| char | char | 1 | 1 |
| string | char a[80] | 80 | 1 |
| varying[n] of char | struct {int, char[n]} | — | 4 |
| record | struct / union | — | 4 |
| array | array | — | Same as element type |
| variant record | struct / union | — | — |
| fields in packed record | Bit Field | — | — |

*Table 7-2* C / Pascal Size and Alignment of Compatible Types with `-xl`

| Pascal Type | C Type | Size (bytes) | Alignment (bytes) |
|---|---|---|---|
| real | float | 4 | 4 |
| integer | short int | 2 | 2 |

## Precautions with Compatible Types

This section describes the precautions to take when working with compatible types.

### The shortrealType

The Pascal `shortreal` and C `float` compatibility works fine if you pass by reference; however, there is some difficulty if you pass by value. For examples of how to do this, see “Value Parameters” on page 99.

### Character Strings

C has the following assumptions about strings:

- All C strings are passed by reference. (C strings are arrays.)
- All C strings are terminated by a null byte.
- All C strings are located in static variable storage.

You can satisfy these assumptions as follows:

- Pass by reference by making the strings `var`, `in`, `out`, or `in out` parameters.
- Provide the null byte explicitly before passing a string to C. This Pascal guarantees the null byte only if the string is a constant. The null byte is not required by the ISO Pascal Standard.

### Array Indexes

Pascal array indexes can start at any integer; C array indexes always start at zero.

### Aggregate Types

Aggregate types include arrays, varying arrays, sets, strings, alphas, records, and variant records.

# 7

Pascal aggregate types may require alignment and layout adjustment to match C unions, structures, and arrays. Pascal aggregate types can be any of the following: arrays, varying arrays, sets, strings, alphas, records, or variant records.

However, you can use the `-calign` option to eliminate some of these differences. With `-calign, the following assumptions are made:`

- Pascal records have the same data layout as C structures.
- Arrays have the same data layout in both languages. However, if you use the `-xl` option in addition to `-calign`, boolean arrays with an odd number of elements are different.
- Pascal variants are the same as C unions.

## *Incompatibilities*

This section describes the incompatibilities between C and Pascal types.

### *Enumerated Types*

C enumerated types are incompatible with Pascal enumerated types. Pascal enumerated types are represented internally by sequences of integral values starting with 0. Storage is allocated for a variable of an enumerated type as if the type were a subrange of integer. For example, an enumerated type of fewer than 128 elements is treated as 0..127, which, according to the rules above, is equivalent to a `char` in C.

C enumerated types are allocated a full word and can take on arbitrary integer values.

### *Pascal Set Types*

In Pascal, a set type is implemented as a bit vector, which may be thought of as a C short-word array, where each short-word contains two bytes. This means that sets are bit-vectors, and they are allocated in multiples of 16 bits. Therefore, the size of a set is as follows:

`ceiling( ord(` *highest_element* `) / 16 )`

Direct access to individual elements of a set is highly machine-dependent and should be avoided. The implementation may change in a future release.

## General Parameter Passing in C and Pascal

A few general rules apply to parameter passing:

- C passes all arrays by reference. C strings are arrays.
- C passes all structures by value.
- In C, if you want to pass anything else by reference, then you must explicitly prepend the reference ampersand (&), or pass an explicit pointer.
- Pascal passes all parameters by value unless you explicitly state that they are `var`, `in out`, or `out` parameters, in which case they are passed by reference.

## C Calls Pascal

This section describes how a C main program calls a Pascal procedure. It consists mainly of examples as follows:

The Pascal procedure, `Samp.p`. Note the procedure definition.

```
procedure samp(var i: integer; var r: real);

begin
    i := 9;
    r := 9.9
end; { samp }
```

The C main program, `Sampmain.c`. Note the procedure definition and call.

```
main ( )
{
   int     i ;
   double  d ;
   extern  samp() ;
   samp( &i, &d ) ;
   printf ( "%d  %3.1f \n", i, d ) ;
}
```

The commands to compile and execute `Samp.p` and `Sampmain.c`:

```
hostname% pc -c Samp.p
hostname% cc Samp.o Sampmain.c -lpc
hostname% a.out
9 9.9
hostname%
```

## *Variable Parameters*

Pascal passes all *variable* parameters by reference, which C can do too.

### *Simple Types without* `-xl`

Without `-xl`, simple types match, as in the following example:

The Pascal procedure, `SimVar.p`:

```
procedure SimVar(var t, f: boolean; var c: char;
                 var i: integer; var r: real;
                 var si: integer16;
                 var sr: shortreal);

begin
    t := true;
    f := false;
    c := 'z';
    i := 9;
    r := 9.9;
    si := 9;
    sr := 9.9
end; { SimVar }
```

The C main program, `SimVarmain.c`:

```
main ( )
{
    char    t, f, c ;
    int     i ;
    double  d ;
    short   si ;
    float   sr ;
    extern  SimVar () ;
    SimVar ( &t, &f, &c, &i, &d, &si, &sr ) ;
    printf ( "%08o %08o %c %d  %3.1f %d %3.1f \n",
                t,   f,   c, i,  d,   si, sr ) ;
}
```

The commands to compile and execute `SimVar.p` and `SimVarmain.c`:

```
hostname% pc -c SimVar.p
hostname% cc SimVar.o SimVarmain.c -lpc
hostname% a.out
00000001 00000000 z 9 9.9 9 9.9
hostname%
```

## *Simple Types with* `-xl`

With `-xl`, the Pascal `real` must be paired with a C `float`, and the Pascal `integer` must be paired with a C `short int`.

## *Strings of Characters*

The C counterpart to the Pascal `alfa` and `string` types are arrays, and C passes all arrays by reference. The C counterpart to the Pascal `varying` is a structure, and C passes structures by value.

Before you call Pascal with a null varying string, set the byte count to zero because that is what Pascal assumes about such strings.

C can pass a structure consisting of a 4-byte integer and an array of characters to a Pascal procedure expecting a `var` parameter that is a variable-length string, as shown in the following example:

The Pascal procedure, `StrVar.p`:

```
type
    varstr = varying [25] of char;

procedure StrVar(var a: alfa; var s: string;
                 var v: varstr);

begin
    a := 'abcdefghi' + chr(0);
    s := 'abcdefghijklmnopqrtstuvwxyz' + chr(0);
    v := 'oyvay' + chr(0)
end; { StrVar }
```

The C main program, `StrVarmain.c`:

```
main ( )
{
    struct VarLenStr {
         int  nbytes ;
         char a[25] ;
    } ;
    struct  VarLenStr vls ;
    char   s10[10], s80[80], s25[25] ;
    extern  StrVar() ;
    vls.nbytes = 0;
    StrVar( s10, s80, &vls ) ;
    strncpy ( s25, vls.a, vls.nbytes ) ;
    printf ( " s10='%s' \n s80='%s' \n s25='%s' \n",
                  s10, s80, s25 ) ;
    printf ( " strlen(s25)=%d \n", strlen(s25) ) ;
}
```

The commands to compile and execute `StrVar.p` and `StrVarmain.c`:

```
hostname% pc -c StrVar.p
hostname% cc StrVar.o StrVarmain.c -lpc
hostname% a.out
 s10='abcdefghi'
 s80='abcdefghijklmnopqrtstuvwxyz'
 s25='oyvay'
 strlen(s25)=5
hostname%
```

## *Fixed Arrays*

For a fixed array parameter, pass the same type and size by reference, as shown in the following example:

The Pascal procedure, `FixVec.p`:

```
type
    VecTyp = array [0..8] of integer;

procedure FixVec(var V: VecTyp; var Sum: integer);

var
    i: integer;

begin
    Sum := 0;
    for i := 0 to 8 do
        Sum := Sum + V[i]
end; { FixVec }
```

The C main program, `FixVecmain.c`:

```
main ()
{
    int    Sum ;
    static int a[] = {1,2,3,4,5,6,7,8,9} ;
    extern FixVec () ;
    FixVec ( a, &Sum ) ;
    printf( " %d \n", Sum ) ;
}
```

The commands to compile and execute `FixVec.p` and `FixVecmain.c`:

```
hostname% pc -c -calign FixVec.p
hostname% cc FixVec.o FixVecmain.c -lpc
hostname% a.out
 45
hostname%
```

Although it does not apply in this example, arrays of aggregates in Pascal have, by default, a size that is always a multiple of four bytes. When you use the `-calign` option to compile the Pascal code, that difference with C is eliminated. The following example illustrates this point. Notice that the string `'Monday'` only gets through to the C main program when you compile the Pascal routine using `-calign`.

The Pascal procedure, `DaysOfWeek.p`:

```
type

  day= array [0..8] of char;
  week = array [0..6] of day;
  year = array [0..51] of week;

procedure DaysOfWeek(var v: year);

begin
  v[1][1] := 'Monday';
end;
```

The C main program, `DaysOfWeekmain.c`:

```
main ( )
{
  char dr[52][7][9] ;
  extern DaysOfWeek() ;
  DaysOfWeek ( dr ) ;
  printf ( "day = '%s' \n", dr[1][1] ) ;

}
```

The commands to compile and execute `DaysOfWeek.p` and `DaysOfWeekmain.c` without `-calign`:

```
hostname% pc -c DaysOfWeek.p
hostname% cc DaysOfWeek.o DaysOfWeekmain.c -lpc
hostname% a.out
day = ''
```

The commands to compile and execute `DaysOfWeek.p` and `DaysOfWeekmain.c` with `-calign`:

```
hostname% pc -c -calign DaysOfWeek.p
hostname% cc DaysOfWeek.o DaysOfWeekmain.c -lpc
hostname% a.out
day = 'Monday '
```

## The `univ` *Arrays*

You can pass any size array to a Pascal procedure expecting a `univ` array, although there is no special gain in doing so, because there is no type/size checking for separate compilations. However, if you want to use an existing Pascal procedure that has a `univ` array, you can do so. All `univ` arrays that are `in`, `out`, `in out`, or `var` parameters pass by reference.

The Pascal procedure, `UniVec.p`, which defines a 10-element array.

```
type
    VecTyp = array [0..9] of integer;

procedure UniVec(var V:univ VecTyp; in Last: integer;
                 var Sum: integer);

var
    i: integer;
begin
    Sum := 0;
    for i := 0 to Last do
        Sum := Sum + V[i]
end; { UniVec }
```

The C main program, `UniVecmain.c`, which passes a 3-element array to the Pascal procedure written to do a 10-element array.

```
main ()
{
    int    Sum ;
    static int a[] = { 7, 8, 9 } ;
    extern UniVec () ;
    UniVec ( a, 2, &Sum ) ;
    printf( " %d \n", Sum ) ;
}
```

The commands to compile and execute `UniVec.p` and `UniVecmain.c`:

```
hostname% pc -c -calign UniVec.p
hostname% cc UniVec.o UniVecmain.c -lpc
hostname% a.out
 24
hostname%
```

## *Conformant Arrays*

For single-dimension arrays, pass *(lower bound, upper bound)* as follows:

```
function ip(var x:array[lb..ub:integer] of real):real;
     ...
     double v1[10];
     extern double ip();
     double z ;
     z = ip ( v1, 0, 9 );
    ...
```

One bounds pair may apply to several arrays if they are declared in the same parameter group:

```
function ip(var x,y:array[lb..ub:integer] of real):real;
      ...
      double v1[10], v2[10] ;
      extern double ip() ;
      double z ;
      z = ip ( v1, v2, 0, 9 ) ;
      ...
```

With multidimensional arrays, for all dimensions but the last, pass (*low bound, high bound, element width).*

Examples of single-dimension, multidimension, and array-of-character conformant arrays follow. Conformant arrays are included here only because they are a relatively standard feature; there are usually more efficient and simpler ways to do the same thing.

### *Example 1: Single-Dimension Array*

The Pascal procedure, `IntCA.p`. Pascal passes the bounds pair.

```
procedure IntCA(var a: array [lb..ub: integer]
                of integer);

begin
    a[1] := 1;
    a[2] := 2
end; { IntCA }
```

The C main program, `IntCAmain.c`:

```
main ()
{
    int k ;
    static int s[] = { 0, 0, 0 } ;
    extern  IntCA () ;
    IntCA ( s, 0, sizeof(s)-1 ) ;
    for (k=0 ; k < 3 ; k++) printf(" %d \n", s[k]) ;
}
```

The commands to compile and execute `IntCA.p` and `IntCAmain.c`:

```
hostname% pc -c -calign IntCA.p
hostname% cc IntCA.o IntCAmain.c -lpc
hostname% a.out
     0
     1
     2
hostname%
```

### *Example 2: Multi-Dimension Array*

The Pascal procedure, `RealCA.p`. Pascal passes (*lowbound, high bound, element width*).

```
procedure RealCA(var A: array [r1..r2: integer] of
                 array [c1..c2: integer] of real);

var
    col, row: integer;

begin
    for row := r1 to r2 do
        for col := c1 to c2 do
            if row = col then
                A[row, col] := 1.0
            else
                A[row, col] := 0.0
end; { RealCA }
```

The C main program, `RealCAmain.c`. Array `M` has 2 rows of 3 columns each. `c1` and `c2` are the first and last columns. `r1` and `r2` are the first and last rows. `wc` is the width of a column element (smallest) and is equal to `sizeof( M[0][0] )`. `wr` is the width of a row element (next largest) and is equal to `(c2-c1+1) * wc`.

```
main ()
{
#define NC 3
#define NR 2
    double M[NR][NC] ;
    int    col, c1, c2, row, r1, r2, wc, wr ;
    extern RealCA() ;
    c1 = r1 = 0 ; c2 = NC-1 ; r2 = NR-1 ;
    wc = sizeof( M[0][0] ) ;
    wr = (c2-c1+1)*wc ;

    RealCA ( M, r1, r2, wr, c1, c2 );
    for ( row = r1 ; row <= r2 ; row++ ) {
       printf("\n");
          for ( col = c1 ; col <= c2 ; col++ )
         printf( "%4.1f ", M[row][col] ) ;
    } ;
    printf("\n");
}
```

The commands to compile and execute `RealCA.p` and `RealCAmain.c`:

```
hostname% pc -c -calign RealCA.p
hostname% cc RealCA.o RealCAmain.c -lpc
hostname% a.out

    1.0  0.0  0.0
    0.0  1.0  0.0
hostname%
```

If `wc` = the width of the smallest element, as determined by `sizeof()`, then the width of the next largest element is the number of those smaller elements in the next larger element multiplied by `wc`.

*width of next largest element* = `(ub - lb + 1) * wc`

In general, (`lb`, `ub`, `wc`) are the bounds and element width of the next lower dimension of the array—this definition is recursive.

### *Example 3: Array of Characters*

The Pascal procedure, `ChrCAVar.p`:

```
procedure ChrCAVar(var a: array [lb..ub: integer]
                  of char);

begin
    a[0] := 'T';
    a[13] := 'o'
end; { ChrCAVar }
```

The C main program, `ChrCAVarmain.c`. For C, the lower bound is always 0.

```
main ()
{
    static char s[] = "this is a string" ;
    extern  ChrCAVar () ;
    ChrCAVar( s, 0, sizeof(s)-1) ; /*(s, lower, upper)*/
    printf( "%11s \n", s ) ;
}
```

The commands to compile and execute `ChrCAVar.p` and `ChrCAVarmain.c`:

```
hostname% pc -c -calign ChrCAVar.p
hostname% cc ChrCAVar.o ChrCAVarmain.c -lpc
hostname% a.out
This is a string
hostname%
```

## *Records/Structures*

In most cases, a Pascal record describes the same objects as its C structure equivalent, provided that the components have compatible types and are declared in the same order. The compatibility of the types depends mostly on size and alignment. For more information on size and alignments of simple components, see Section , “Compatibility of Types for C and Pascal,” on page 78.

By default, the alignment of a record is always four bytes and the size of a record is always a multiple of four bytes. However, when you use `-calign` in compiling the Pascal code, the size and alignment of the Pascal record matches the size and alignment of the equivalent C structure.

A Pascal record of an integer and a character string matches a C structure of the same things as follows:

The Pascal procedure, `StruChr.p`. It is safer for the Pascal procedure to explicitly provide the null byte and include it in the count before the string is passed to C.

```
type
    lenstr =
      record
          nbytes: integer;
          chrstr: array [0..24] of char
      end;

procedure StruChr(var v: lenstr);

begin
    v.chrstr := 'water' + chr(0);
    v.nbytes := 6
end; { StruChr }
```

The C main program, `StruChrmain.c`:

```
main ( )
{
    struct VarLenStr {
         int  nbytes ;
         char a[25] ;
    } ;
    struct  VarLenStr vls ;
    char   s25[25] ;
    extern  StruChr() ;
    vls.nbytes = 0;
    StruChr ( &vls ) ;
    strncpy ( s25, vls.a, vls.nbytes ) ;
    printf ( " s25='%s' \n", s25 ) ;
    printf ( " strlen(s25)=%d \n", strlen(s25) ) ;
}
```

The commands to compile and execute `StruChr.p` and `StruChrmain.c`:

```
hostname% pc -c StruChr.p
hostname% cc StruChr.o StruChrmain.c -lpc
hostname% a.out
 s25='oyvay'
 strlen(s25)=5
hostname%
```

The record in the example above has, by default, the same size and alignment as the equivalent C record. Some records, though, will be laid out differently unless you use the `-calign` option. Consider this example:

The Pascal routine, `DayWeather.p`:

```
type
    str = record
            day: array [0..8] of char;
            weather: array [0..20] of char;
            end;
    strarray = array [0..1] of str;

procedure StruChr(var v: strarray; var weathersiz: integer);
begin
    v[1].day := 'Monday';
    v[1].weather := 'sunny';
    weathersiz := 5;
end; { StruChr }
```

The C main program, `DayWeathermain.c`:

```
main ( )
    {
    struct dayrec {
            char date[9] ;
            char weather[21] ;
    } ;
    struct dayrec dr[2] ;
    int nbytes;
    char s25[25] ;
    char t25[25];
    extern StruChr() ;
    nbytes = 0;
    StruChr ( dr, &nbytes ) ;
    strncpy ( s25, dr[1].date, 6 ) ;
    printf ( " day = '%s' \n", s25 ) ;
    strncpy ( t25, dr[1].weather, nbytes ) ;
    printf ( " weather = '%s' \n", t25 ) ;

}
```

When you compile the Pascal routine without using the `-calign` option, the program does not work correctly.

The commands to compile and execute `DayWeather.p` and `DayWeathermain.c` without `-calign`:

```
hostname% pc -c DayWeather.p
hostname% cc DayWeather.o DayWeathermain.c -lpc
hostname% a.out
 day = ''
 weather = ' sun'
```

The commands to compile and execute `DayWeather.p` and `DayWeathermain.c` with `-calign`:

```
hostname% pc -calign -c DayWeather.p
hostname% cc DayWeather.o DayWeathermain.c -lpc
hostname% a.out
 day = 'Monday'
 weather = 'sunny'
```

## *Variant Records*

C equivalents of variant records can sometimes be constructed, although there is some variation with architecture, and sometimes a need to adjust alignment. You can avoid the need to adjust alignment by using the `-calign` option.

The Pascal procedure, `VarRec.p`:

```
type
    vr =
      record
          case tag: char of
              'a': ( ch1, ch2: char );
              'b': ( flag: boolean );
              'K': ( ALIGN: integer )
      end;

procedure VarRec(var x: vr);

begin
    if x.ch1 = 'a' then
        x.ch2 := 'Z'
end; { VarRec }
```

The C main program, `VarRecmain.c`:

```
main ()
{
    struct vlr {
        char tag;
        union {
            struct {
                char ch1, ch2;
            }a_var;
            struct {
                char flag;
            }b_var;
            struct {
                int ALIGN;
            }c_var;
        }var_part;
    } *x;
    x = (struct vlr*)malloc(sizeof(struct vlr)) ;
    x->tag = 'a'  ;
    x->var_part.a_var.ch1 = 'a'  ;
    x->var_part.a_var.ch2 = 'b'  ;
    VarRec ( x ) ;
    printf( "%c\n", x -> var_part.a_var.ch2 ) ;
}
```

The commands to compile and execute `VarRec.p` and `VarRecmain.c`.

```
hostname% pc -c -calign VarRec.p
hostname% cc VarRec.o VarRecmain.c -lpc
hostname% a.out
Z
hostname%
```

## *Pascal Set Type*

In Pascal, a set type is implemented as a bit vector, which may be thought of as a C short-word array. Direct access to individual elements of a set is highly machine-dependent and should be avoided. The implementation may change in a future release.

In Pascal, bits are numbered within a byte from the most significant to least, as shown in Table 7-3.

*Table 7-3* Set Implementation

| Set | Bit Numbering |
|---|---|
| set+3: | 31, 30, 29, 28, 27, 26, 25, 24 |
| set+2: | 23, 22, 21, 20, 19, 18, 17, 16 |
| set+1: | 15, 14, 13, 12, 11, 10, 9, 8 |
| set+0: | 7, 6, 5, 4, 3, 2, 1, 0 |

In C, a set could be described as a short-word array beginning at an even address. Also note that with the current set representation, it does not matter what the lower-bound value is.

The *n*th element in a set `[lower...upper]` can be tested as follows:

```
#define LG2BITSLONG 5 /* log2( bits in long word) */
#define LG2BITSWORD 4 /* log2( bits in short word) */
#define MSKBITSLONG 0x1f
#define MSKBITSHORT 0x0f

    short *setptr; /* set as array of shorts */
    int upper;     /* upper bound of the set */
    int elem;      /* ordinal value of set element */
    int i;

    if  ( setptr[elem >> LG2BITSWORD]  &
    (1 <<  (elem & MSKBITSWORD))      )  {
              /* elem is in set */
    }
```

## *Pascal* `intset` *Type*

The Pascal `intset` type is predefined as `set of [0..127]`. A variable of this type takes 16 bytes of storage.

The Pascal procedure, `IntSetVar.p`, which has an `intset` of the elements [1, 3, 7, 8]:

```
procedure IntSetVar(var s: intset);
begin
    s := [1, 3, 7, 8]
end; { IntSetVar }
```

The C main program, `IntSetVarmain.c`:

```
main ()
{
    int  w ;
    unsigned int  *p, *s ;
    extern  IntSetVar() ;
    s = (unsigned int *) malloc(16) ;

    IntSetVar ( s ) ;

    for ( w=0, p=s ; w < 4 ; w++, p++ )
        printf( "%012o %3d \n", *p, w ) ;
    printf("110 001 010 (binary, word 4)\n") ;
    printf("876 543 210 (bits, word 4)") ;
    printf("\n") ;
}
```

The commands to compile and execute `IntSetVar.p` and `IntSetVarmain.c`. The output of this example depends on the architecture of your machine. This example was run on a Sun-4 workstation.

```
hostname% pc -c IntSetVar.p
hostname% cc IntSetVar.o IntSetVarmain.c -lpc
hostname% a.out
000000000000 0
000000000000 1
000000000000 2
000000000612 3
110 001 010 (binary, word 4)
876 543 210 (bits, word 4)
hostname%
```

## *Value Parameters*

### *Simple Types without* `-xl`

Without `-xl`, simple types match, as in the following example:

The Pascal procedure, `SimVal.p`. `t`, `f`, `c`, `i`, `r`, and `s` are value parameters.

```
procedure SimVal(t, f: boolean; c: char; i: integer;
                 r: real; s: integer16;
                 var reply: integer);

begin
    reply := 0;
    {If nth arg ok, set nth octal digit to one.}
    if t then
        reply := reply + 1;
    if not f then
        reply := reply + 8;
    if c = 'z' then
        reply := reply + 64;
    if i = 9 then
        reply := reply + 512;
    if r = 9.9 then
        reply := reply + 4096;
    if s = 9 then
        reply := reply + 32768
end; { SimVal }
```

The C main program, `SimValmain.c`:

```
main ( )
{
    char   t = 1, f = 0 ;
    char   c = 'z' ;
    int    i = 9 ;
    double d = 9.9 ;
    short  s = 9 ;
    int    args ;
    extern  SimVal() ;

    SimVal( t, f, c, i, d, s, &args ) ;

    printf("args=%06o (If nth digit=1, arg n OK)\n",
        args ) ;
}
```

The commands to compile and execute `SimVal.p` and `SimValmain.c`:

```
hostname% pc -c SimVal.p
hostname% cc SimVal.o SimValmain.c -lpc
hostname% a.out
args=111111 (If nth digit=1, arg n OK)
hostname%
```

### *Simple Types with* `-xl`

With `-xl`, the Pascal `real` must be paired with a C `float`, and the Pascal `integer` must be paired with a C `short int`. For coping with a C value `float` parameter, see the next section, “Type shortreal.

## *Type* `shortreal`

There is no straightforward way to pass `shortreal` value parameters between Pascal and C, because Pascal passes such parameters in single precision. In SPARCompiler C, the default action is to promote `float` parameters to `double` before pushing them onto the stack.

However, if a Pascal procedure with a `shortreal` value parameter must be called from C, you can try the option which causes *all* single-precision parameters of C functions to be passed as single-precision, rather than as double precision: the `-fsingle2` option of the `cc` command.

The option `-fsingle2` makes Pascal `shortreal` parameters match C `float` parameters. This is a trick and not a general method. It requires unusual caution and is not portable.

---

**Note –** Do not mix C modules compiled with `-fsingle2` and C modules compiled without it. That is, you must recompile all C modules with the `-fsingle2` option for this to work.

---

The Pascal procedure, `ShortRealVal.p`:

```
procedure ShortRealVal(sr: shortreal; var r: real);

begin
    r := sr + 1.0
end; { ShortRealVal }
```

The C main program, `ShortRealValmain.c`. The C `float` parameter `f` is successfully matched to the Pascal `shortreal` parameter `sr`.

```
main ( )
{
    float   f = 2.0 ;
    double  d ;
    extern  ShortRealVal () ;

    ShortRealVal ( f, &d ) ;

    printf("%3.1f %3.1f \n", (double)f, d) ;
}
```

The commands to compile and execute `ShortRealVal.p` and `ShortRealValmain.c`. The **`-fsingle2`** option necessitates casting `f` to `double` in the `printf` statement.

```
hostname% pc -c ShortRealVal.p
hostname% cc -fsingle2 ShortRealVal.o ShortRealValmain.c -lpc
hostname% a.out
2.0 3.0
hostname%
```

## *Arrays*

Since C cannot pass arrays by *value*, it cannot pass strings of characters, fixed arrays, or `univ` arrays by value.

## *Conformant Arrays*

Pascal passes all *value* parameters on the stack or in registers with the exception of value conformant array parameters, which are handled by creating a copy in the caller environment and passing a pointer to the copy. In addition, the bounds of the array must be passed (see “Conformant Arrays” on page 88).

This example is the same as the single-dimension example in “Conformant Arrays, except that the `var` prefix is deleted.

The Pascal procedure, `ChrCAVal.p`:

```
procedure ChrCAVal(a: array [lb..ub: integer] of char);

begin
    a[0] := 'T';
    a[13] := 'o'
end; { ChrCAVal }
```

The C main program, `ChrCAValmain.c`:

```
main ()
{
    static char s[] = "this is a string" ;
    extern  ChrCAVal () ;
    ChrCAVal( s, 0, sizeof(s)-1) ; /*(s, lower, upper)*/
    printf( "%11s \n", s ) ;
}
```

The commands to compile and execute `ChrCAVal.p` and `ChrCAValmain.c`:

```
hostname% pc -c -calign ChrCAVal.p
hostname% cc ChrCAVal.o ChrCAValmain.c -lpc
hostname% a.out
This is a string
hostname%
```

## *Function Return Values*

Function return values match types the same as with parameters, and they pass in much the same way.

### *Simple Types*

The simple types pass in a straightforward way as follows:

The Pascal function, `RetReal.p`:

```
function RetReal(x: real): real;

begin
    RetReal := x + 1
end; { RetReal }
```

The C main program, `RetRealmain.c`:

```
main ( )
{
    float r, s ;
    extern  float RetReal() ;
    r = 2.0 ;
    s = RetReal( r ) ;
    printf( "%f\n", s ) ;
}
```

The commands to compile and execute `RetReal.p` and `RetRealmain.c`:

```
hostname% pc -c RetReal.p
hostname% cc RetReal.o RetRealmain.c -lpc
hostname% a.out
3.000000
hostname%
```

## *Type* `shortreal`

To return a `shortreal` as a function value, use special macros which you will find in the standard `#include` file `math.h`:

The Pascal function, `RetShortReal.p`:

```
function RetShortReal(x: real): shortreal;

begin
    RetShortReal := x + 1.0
end; { RetShortReal }
```

The C main program, `RetShortRealmain.c`:

```
include <math.h>
main ( )
{
    float r, s ;
    extern  FLOATFUNCTIONTYPE RetShortReal() ;
    r = 2.0 ;
    ASSIGNFLOAT( s, RetShortReal( r ) ) ;
    printf( "%8.6f\n", s ) ;
}
```

The commands to compile and execute `RetShortReal.p` and `RetShortRealmain.c`:

```
hostname% pc -c RetShortReal.p
hostname% cc RetShortReal.o RetShortRealmain.c -lpc
hostname% a.out
3.000000
hostname%
```

### *Input and Output*

If your C main program calls a Pascal procedure that does I/O, then include the following code in the C main before you call the Pascal procedure:

```
__PC0__PCSTART();
```

Also, in the C main program just before exit, add the following:

```
__PC0__PCEXIT();
```

## *Pascal Calls C*

This section parallels the section Section , “C Calls Pascal,” on page 81. Earlier comments and restrictions apply here, too.

### *Variable Parameters*

Pascal passes all *variable* parameters by reference, which C can do, too.

#### *Simple Types*

Simple types pass in a straightforward manner as follows:

The C function, `SimVar.c`:

```
SimVar ( t, f, c, i, d, si, sr )
    char    * t, * f, * c ;
    int     * i ;
    double  * d ;
    short   * si ;
    float   * sr ;
{
    *t = 1 ; *f = 0 ;
    *c  = 'z' ;
    *i  = 9 ;
    *d  = 9.9 ;
    *si = 9 ;
    *sr = 9.9 ;
}
```

The Pascal main program, `SimVarmain.p`:

```
program SimVarmain(output);

var
    t, f: boolean;
    c: char;
    i: integer;
    r: real;
    si: integer16;
    sr: shortreal;

procedure SimVar(var t, f: boolean; var c: char;
                 var i: integer; var r: real;
                 var si: integer16;
                 var sr: shortreal); external c;

begin
    SimVar(t, f, c, i, r, si, sr);
    writeln(t, f: 6, c: 2, i: 2, r: 4: 1, si: 2, sr: 4: 1)
end. { SimVarmain }
```

The commands to compile and execute `SimVar.c` and `SimVarmain.p`:

```
hostname% cc -c SimVar.c
hostname% pc SimVar.o SimVarmain.p
hostname% a.out
true false z 9 9.9 9 9.9
hostname%
```

## *Strings of Characters*

The `alfa` and `string` types pass simply; varying strings are a little tricky. All pass by reference.

The C function, `StrVar.c`:

```
StrVar ( s10, s80, vls )
    char   *s10, *s80 ;
    struct VarLenStr {
         int  nbytes ;
         char a[26] ;
    } *vls ;
{
    static char ax[11] = "abcdefghij" ;
    static char sx[81] = "abcdefghijklmnopqrtstuvwxyz" ;
    static char vx[6]  = "oyvay" ;
    strncpy ( s10, ax, 11 ) ;
    strncpy ( s80, sx, 80 ) ;
    strncpy ( vls -> a, vx, 5 ) ;
    vls -> nbytes =  5 ;
}
```

The Pascal main program, `StrVarmain.p`:

```
program StrVarmain(output);

type
    varstr = varying [26] of char;

var
    a: alfa;
    s: string;
    v: varstr;

procedure StrVar(var xa: alfa; var xs: string;
                 var xv: varstr); external c;

begin
    StrVar(a, s, v);
    writeln(a);
    writeln(s);
    writeln(v);
    writeln('length(v)= ', length(v): 2)
end. { StrVarmain }
```

The commands to compile and execute `StrVar.c` and `StrVarmain.p`:

```
hostname% cc -c StrVar.c
hostname% pc StrVar.o StrVarmain.p
hostname% a.out
abcdefghij
abcdefghijklmnopqrtstuvwxyz
oyvay
length(v)= 5
hostname%
```

Avoid constructs that rely on strings being in static variable storage. For example, you could use `mktemp`(3) in Pascal as follows:

Incorrect use of string in static variable storage.

```
tmp := mktemp('/tmp/eph.xxxxxx')
```

This is incorrect, since `mktemp()` modifies its argument. A correct solution is to use the C library routine `strncpy()` (see `string`(3)) to copy the string constant to a declared `char` array variable:

Correct solution using the C library routine `strncpy()`.

```
program Use_mktemp ;

procedure strncpy( var dest: univ string ;
    var srce: univ string ;
    length: integer) ;  external c ;

procedure mktemp(var dest: univ string); external c;
    ...
var path: string ;
begin
    ...
strncpy( path, '/tmp/eph.xxxxxx', sizeof(path)) ;
mktemp( path ) ;
    ...
end .
```

## *Fixed Arrays*

For a fixed-array parameter, pass the same type and size:

The C function, `FixVec.c`:

```
FixVec ( V, Sum )
    int *Sum ;
    int V[9] ;
{
    int i ;
    *Sum = 0 ;
    for ( i = 0; i <= 8 ; i++ ) *Sum = *Sum + V[i] ;
}
```

The Pascal main program, `FixVecmain.p`:

```
program FixVecmain(output);

type
    VecTyp = array [0..8] of integer;

var
    V: VecTyp := [1, 2, 3, 4, 5, 6, 7, 8, 9];
    Sum: integer;

procedure FixVec(var XV: VecTyp; var XSum: integer);
                external c;

begin
    FixVec(V, Sum);
    writeln(Sum: 3)
end. { FixVecmain }
```

The commands to compile and execute `FixVec.c` and `FixVecmain.p`:

```
hostname% cc -c FixVec.c
hostname% pc -calign FixVec.o FixVecmain.p
hostname% a.out
 45
hostname%
```

The `-calign` option isn't needed for this example, but might be needed if the array parameter was an array of aggregates.

## The `univ` *Arrays*

The `univ` arrays that are `in`, `out`, `in out`, or `var` parameters pass by reference.

The C function, `UniVec.c`:

```
void UniVec(V, Last, Sum)
    int V[3] ;
    int Last ;
    int *Sum ;
{
    int i ;
    *Sum = 0 ;
    for (i=0;i<=Last;i++) *Sum += V[i];
}
```

The Pascal main program, `UniVecmain.p`:

```
program UniVecPcC(output);

type
    VecTyp = array [0..9] of integer;

procedure UniVec(var V:univ VecTyp; in Last: integer;
                 var Sum: integer); external c;

var
    Sum: integer;
    V: array [0..2] of integer;

begin
    V[0] := 7;
    V[1] := 8;
    V[2] := 9;
    UniVec(V, 2, Sum);
    writeln(Sum)
end. { UniVecPcC }
```

The commands to compile and execute `UniVec.c` and `UniVecmain.p`:

```
hostname% cc -c UniVec.c
hostname% pc -calign UniVec.o UniVecmain.p
hostname% a.out
          24
hostname%
```

The `-calign` option isn't needed for this example, but might be needed if the array parameter was an array of aggregates.

## *Conformant Arrays*

For single-dimension conformant arrays, pass upper and lower bounds, placed after the declared parameter list. If the array is multidimensional, pass element widths as well, one element width for each dimension except the last. Chapter 8, "The FORTRAN–Pascal Interface has an example of multidimensional conformant array passing.

The following example uses a single-dimension array:

The C function, `IntCA.c`:

```
void IntCA ( a, lb, ub )
int a[] ;
int lb, ub ;
{
    int k ;
    for ( k=0 ; k <= ub-lb ; k++ ) a[k]=4 ;
} ;
```

The Pascal main program, `IntCAmain.p`. Note that what Pascal passes as `s`, is received in C as `a, lb, ub`.

```
program IntCAmain(output);

procedure IntCA(var a: array [lb..ub: integer]
                of integer); external c;

var
    s: array [1..3] of integer;
    i: integer;

begin
    IntCA(s);
    for i := 1 to 3 do
      write(s[i]);
    writeln
end. { IntCAmain }
```

The commands to compile and execute `IntCA.c` and `IntCAmain.p`:

```
hostname% cc -c IntCA.c
hostname% pc -calign IntCA.o IntCAmain.p
hostname% a.out
    4               44
hostname%
```

The `-calign` option isn't needed for this example, but might be needed if the array parameter was an array of aggregates.

### *Records/Structures*

In most cases, a Pascal record describes the same objects as its C structure equivalent, provided that the components have compatible types and are declared in the same order. For more information, see Section , "Compatibility of Types for C and Pascal," on page 78.

Records that contain aggregates may differ because aggregates in C and Pascal sometimes have different sizes and alignments. If you compile the Pascal code with the `-calign` option, the differences are eliminated.

A Pascal record of an integer and a character string matches a C structure of an integer and an array of `char` values as follows:

The C function, `StruChr.c`:

```
struct VarLenStr {
    int  nbytes ;
    char a[26] ;
} ;
void StruChr ( v )
struct  VarLenStr *v ;
{
    bcopy( "oyvay", v -> a, 5 ) ;
    v -> nbytes = 5 ;
}
```

The Pascal main program, `StruChrmain.p`:

```
program StruChrmain;

type
    VarLenStr =
      record
          nbytes: integer;
          a: array [0..25] of char
      end;

var
    vls: VarLenStr;
    i: integer;

procedure StruChr(var vls: VarLenStr); external c;

begin
    StruChr(vls);
    write('string=''');
    for i := 0 to vls.nbytes - 1 do
        write(vls.a[i]);
    writeln('''');
    writeln('strlen=', vls.nbytes)
end. { StruChrmain }
```

The commands to compile and execute `StruChr.c` and `StruChrmain.p`:

```
hostname% cc -c StruChr.c
hostname% pc -calign StruChr.o StruChrmain.p
hostname% a.out
string='oyvay'
strlen=          5
hostname%
```

## *Variant Records*

C equivalents of variant records can sometimes be constructed, although there is some variation with architecture, and sometimes a need to adjust alignment.

The C function, `VarRec.c`:

```
VarRec ( x )
        struct {
                char tag;
                union {
                        struct {
                                char ch1, ch2;
                        }a_var;
                        struct {
                                char flag;
                        }b_var;
                        struct {
                                int ALIGN;
                        }c_var;
                }var_part;
        } *x;
{
      if ( x->var_part.a_var.ch1 == 'a' )
              x->var_part.a_var.ch2 = 'Z' ;
}
```

The Pascal main program, `VarRecmain.p`:

```
program VarRecmain;
type
    vr =
      record
          case tag: char of
              'a': ( ch1, ch2: char );
              'b': ( flag: boolean );
              'K': ( ALIGN: integer )
      end;

var
    x: vr;

procedure VarRec(var d: vr); external c;

begin
    x.tag := 'a';
    x.ch1 := 'a';
    x.ch2 := 'b';
    VarRec(x);
    writeln(x.ch2)
end. { VarRecmain }
```

The commands to compile and execute `VarRec.c` and `VarRecmain.p`:

```
hostname% cc -c VarRec.c
hostname% pc -calign VarRec.o VarRecmain.p
hostname% a.out
Z
hostname%
```

The `-calign` option isn't needed for the previous example, but might be needed if the record contained aggregates.

## *Non-Pascal Procedures*

When you use the `-xl` option in compiling Pascal code, you can use the `nonpascal` keyword to declare that an external procedure is written in another language. This keyword generally causes everything to be passed by reference.

The C function, `NonPas.c`. In the function `for_C`, `s` is a pointer (declared `var` in the procedure declaration), and `len` is not a pointer (not declared `var` in the procedure declaration). In the function `for_nonpascal`, `s` is still a pointer (though not declared `var` in the procedure declaration), and `len` is now a pointer (though not declared `var`).

```
for_C ( s, len )
    char      *s ; /* a ptr, a var in the call */
    int       len ; /* a value, not a var in the call */
{
    int       i ;
    for ( i = 0 ; i < len ; i++ )
      putchar( s[i] ) ;
    putchar('\n') ;
}
for_nonpascal ( s, len )
    char      *s ; /* a ptr, though not var in the call */
    int       *len ; /* a ptr, though not var in the call */
{
    int       i ;
    for ( i = 0 ; i < *len ; i++ )
          putchar( s[i] ) ;
      putchar('\n') ;
}
```

The Pascal main program, `NonPasmain.p`:

```
program NonPasmain;

var
    s: string;

procedure for_C(var s: string; len: integer);
                external c;

procedure for_nonpascal(s: string; len: integer);
                        nonpascal;

begin
    s := 'Hello from Pascal';
    for_C(s, 17);
    for_nonpascal(s, 17)
end. { NonPasmain }
```

The commands to compile and execute `NonPas.c` and `NonPasmain.p`:

```
hostname% cc -c NonPas.c
hostname% pc NonPas.o NonPasmain.p
hostname% a.out
Hello from Pascal
Hello from Pascal
hostname%
```

## *Value Parameters*

In general, Pascal passes *value* parameters in registers or on the stack, widening to a full word if necessary.

### *Simple Types*

With *value* parameters, simple types match as in the following example:

The C function, `SimVal.c`:

```
SimVal( t, f, c, i, d, s, reply )
    char   t, f ;
    char   c  ;
    int    i ;
    double d ;
    short  s ;
    int    * reply ;
{
    *reply = 0 ;
    /* If nth arg ok, set nth octal digit to one */
    if ( t        ) *reply = *reply + 1 ;
    if ( ! f      ) *reply = *reply + 8 ;
    if ( c == 'z' ) *reply = *reply + 64 ;
    if ( i == 9   ) *reply = *reply + 512 ;
    if ( d == 9.9 ) *reply = *reply + 4096 ;
    if ( s == 9   ) *reply = *reply + 32768 ;
}
```

The Pascal main program, `SimValmain.p`:

```
program SimVal(output);

var
    t: boolean := true;
    f: boolean := false;
    c: char := 'z';
    i: integer := 9;
    r: real := 9.9;
    s: integer16 := 9;
    args: integer;

procedure SimVal(t, f: boolean; c: char; i: integer;
                 r: real; s: integer16; var reply: integer);
                external c;

begin
    SimVal(t, f, c, i, r, s, args);
    writeln('args=', args: 6 oct, ' (If nth digit=1, arg n OK.)')
end. { SimVal }
```

The commands to compile and execute `SimVal.c` and `SimValmain.p`:

```
hostname% cc -c SimVal.c
hostname% pc SimVal.o SimValmain.p
hostname% a.out
args=111111 (If nth digit=1, arg n OK.)
hostname%
```

## *Type* `shortreal`

If Pascal calls C, there are two ways to pass `shortreal` parameters. For more on `float` parameters, see Section , “C Calls Pascal,” on page 81.

***Example 1:***

The C function, `ShortRealVal.c`:

```
ShortRealVal ( f, d )
    float   f ;
    double  * d ;
{
    *d = f + 1.0 ;
}
```

The Pascal main program, `ShortRealValmain.p`:

```
program ShortRealValmain(output);

var
    sr: shortreal := 2.0;
    r: real;

procedure ShortRealVal(xsh: shortreal; var xr: real);
                        external c;

begin
    ShortRealVal(sr, r);
    writeln(sr: 4: 1, r: 4: 1)
end. { ShortRealValmain }
```

The commands to compile and execute `ShortRealVal.c` and `ShortRealValmain.p`:

```
hostname% cc -c ShortRealVal.c
hostname% pc ShortRealVal.o ShortRealValmain.p
hostname% a.out
 2.0 3.0
hostname%
```

***Example 2:***

The C function, `ShortRealVal2.c`:

```
ShortRealVal2 ( f, d )
    float   f ;
    double  * d ;
{
    *d = f + 1.0 ;
}
```

The Pascal main program, `ShortRealVal2main.p`:

```
program ShortRealVal2main(output);

var
    sr: shortreal := 2.0;
    r: real;

procedure ShortRealVal2(xsh: shortreal; var xr: real);
                        external;

begin
    ShortRealVal2(sr, r);
    writeln(sr: 4: 1, r: 4: 1)
end. { ShortRealVal2main }
```

The commands to compile and execute are: `ShortRealVal2.c` and `ShortRealVal2main.p`:

```
hostname% cc -fsingle2 -c ShortRealVal2.c
hostname% pc ShortRealVal2.o ShortRealVal2main.p
hostname% a.out
 2.0 3.0
hostname%
```

## Function Return Values

Function return values match types the same as with parameters, and they pass in much the same way. See “Variable Parameters” on page 82.

The following example shows how to pass simple types:

The C function, `RetReal.c`:

```
float RetReal ( x )
    float *x ;
{
    return ( *x + 1.0 ) ;
}
```

The Pascal main program, `RetRealmain.p`:

```
program RetRealmain;

var
    r, s: real;

function RetReal(var x: real): real;
                external c;

begin
    r := 2.0;
    s := RetReal(r);
    writeln(r: 4: 1, s: 4: 1)
end. { RetRealmain }
```

The commands to compile and execute `RetReal.c` and `RetRealmain.p`:

```
hostname% cc -c RetReal.c
hostname% pc RetReal.o RetRealmain.p
hostname% a.out
 2.0 3.0
hostname%
```

## Parameters That Are Pointers to Procedures

Pascal has a special type that is a pointer to a procedure. A variable of this type can be used as a parameter:

The C function, `ProcPar.c`:

```
void proc_c ( p )
    void (*p)() ;     /* a pointer to procedure argument */
{
    char *s ;
    s = "Called from C" ;
    (*p)( s, strlen(s)) ;     /* Call the Pascal routine */
}
```

The Pascal main program, `ProcParmain.p`, which calls the C procedure `proc_c`, passing it the address of the Pascal procedure `proc_pas`. The C procedure assigns a value to the string `s`, and calls the procedure whose pointer it just received. Then the Pascal procedure `proc_pas` writes a literal constant and the string it just received.

```
program ProcParmain;

type
    { Declare a procedure pointer type. }
    proc_ptr = ^ procedure(var s: string; i: integer);

{ Declare an external C procedure
  which takes a procedure argument. }

procedure proc_c(p: proc_ptr); external c;

procedure proc_pas(var cstr: string; strlen: integer);

var
    i: integer;

begin
    write('Hello from PROC_PASCAL: ');
    for i := 1 to strlen do begin
        write(cstr[i])
    end;
    writeln
end; { proc_pas }

begin
    { Call the C routine. }
    proc_c(addr(proc_pas))
end. { ProcParmain }
```

The commands to compile and execute `ProcPar.c` and `ProcParmain.p`:

```
hostname% cc -c ProcPar.c
hostname% pc ProcPar.o ProcParmain.p
hostname% a.out
Hello from PROC_PASCAL: Called from C
hostname%
```

## *Procedures and Functions as Parameters*

It is probably clearer to pass a pointer to a procedure than to pass the procedure name itself. See Section , “Pascal Calls C,” on page 105.

A procedure or function passed as an argument is associated with a static link to its lexical parent’s activation record. When an outer block procedure or function is passed as an argument, Pascal passes a null pointer in the position normally occupied by the passed routine’s static link. So that procedures and functions can be passed to other languages as arguments, the static links for all procedure or function arguments are placed after the end of the conformant array bounds pairs (if any).

Routines in other languages may be passed to Pascal; a dummy argument must be passed in the position normally occupied by the passed routine’s static link. If the passed routine is not a Pascal routine, the argument is used only as a place holder.

## *Global Variables in C and Pascal*

If the types are compatible, a global variable can be shared between C and Pascal:

The Pascal procedure, `GloVar.p`:

```
var
    stuff: integer;

procedure GloVar;

begin
    stuff := 9
end; { GloVar }
```

The C main program, `GloVarmain.c`:

```
int stuff ;
main ()
{
    extern GloVar () ;
    stuff = 3 ;
    GloVar ( ) ;
    printf( " %d \n", stuff ) ;
}
```

The commands to compile and execute `GloVar.p` and `GloVarmain.c` without `-xl`. With `-xl`, the Pascal `integer` must be paired with a C `short int`, and would have to be declared public as the default visibility is private.

```
hostname% pc -c GloVar.p
hostname% cc GloVar.o GloVarmain.c
hostname% a.out
 9
hostname%
```

## *Passing Files Between Pascal and C*

You can pass a file pointer from Pascal to C, then have C do the I/O.

The C procedure, `UseFilePtr.c`:

```
UseFilePtr ( ptr )
    int *ptr ;
{ /* Write to the file: */
     fprintf( ptr, "[1] Passing the file descriptor \n") ;
    fprintf( ptr, "[2] and writing information \n") ;
    fprintf( ptr, "[3] to a file \n") ;
}
```

The Pascal main program, `UseFilePtrmain.p`:

```
program UseFilePtrmain;

var
    f: text;
    cfile: univ_ptr;

procedure UseFilePtr(cf: univ_ptr); external c;

begin
    rewrite(f, 'myfile.data'); { Make the file. }
    cfile := getfile(f);      { Get a file pointer. }
    UseFilePtr(cfile)         { Call the C function. }
end. { UseFilePtrmain }
```

The commands to compile and execute `UseFilePtc.c` and `UseFilePtrmain.p`:

```
hostname% cc -c UseFilePtr.c
hostname% pc UseFilePtr.o UseFilePtrmain.p
hostname% a.out
hostname% cat myfile.data
[1] Passing the file descriptor
[2] and writing information
[3] to a file
hostname%
```

# *The FORTRAN–Pascal Interface* 8

This chapter describes how to mix FORTRAN and Pascal modules in the same program. The FORTRAN programs that appear in this chapter are written with Sun FORTRAN Release 1.2.

## *Compiling Mixed-Language Programs*

The compiler driver automatically brings in the runtime libraries for the main module. (Compile your program with the `-v` [verbose] option to see.) However, when you compile a module that is not the main module, and which is written in a language different than the main module, you must explicitly bring in the runtime library on the command line.

For example, you must use the compiler options `-lpfc` and `-lpc` when you compile a FORTRAN main routine that calls Pascal. The `-lpfc` option links the common startup code for programs containing mixed Pascal and FORTRAN object libraries. The `-lpc` option includes the Pascal object library `libpc`.

You must specify `-lpfc` on the command line before `-lpc`. For example:

```
hostname% pc -c my_pascal.p
hostname% f77 my_pascal.o my_fortran.f -lpfc -lpc
Sampmain.f:
 MAIN:
hostname%
```

The `-c` option to `pc` produces an unlinked object file.

When you compile a Pascal main routine that calls FORTRAN you must use the compiler options `–lpfc` and `–lF77`. The `–lF77` option links the FORTRAN object library `libf77`.

You must specify `–lpfc` on the command line before `–lF77`. For example:

```
hostname% f77 -c my_fortran.f
hostname% pc my_fortran.o my_pascal.p -lpfc -lF77
my_fortran.f:
     MAIN:
hostname%
```

You can omit the libraries if the foreign language module does not interact with the runtime environment (does no I/O, memory allocation, and so on). However, there is no overhead to linking to an unused library; therefore, always link in the appropriate runtime libraries, even if you think you might not need them.

## *Compatibility of Types for FORTRAN and Pascal*

Tables 8-1 lists the default sizes and alignments of compatible types for FORTRAN and Pascal.

*Table 8-1* Default Size and Alignment of Compatible Types in Pascal and FORTRAN

| Pascal Type | FORTRAN Type | Size (bytes) | Alignment (bytes) |
|---|---|---|---|
| double | double precision | 8 | 8 |
| longreal | double precision | 8 | 8 |
| real | double precision | 8 | 8 |
| single | real | 4 | 4 |
| shortreal | real | 4 | 4 |
| integer16 | integer*2 | 2 | 2 |
| integer32 | integer*4 | 4 | 4 |
| integer | integer*4 | 4 | 4 |
| -128..127 | logical*1, byte, or character | 1 | 1 |

*Table 8-1* Default Size and Alignment of Compatible Types in Pascal and FORTRAN

| Pascal Type | FORTRAN Type | Size (bytes) | Alignment (bytes) |
|---|---|---|---|
| boolean | logical*1, byte, or character | 1 | 1 |
| alfa | character*10 | 10 | 1 |
| char | character | 1 | 1 |
| string | character*80 | 80 | 1 |
| varying[n] of char | structure /v/ integer*4 character*n end structure | — | 4 |
| array | array | Same as element type | |
| record | structure | — | 4 |

Tables 8-2 lists the default sizes and alignments of compatible types for FORTRAN and Pascal with the `-xl` option:

*Table 8-2* Size and Alignment of Compatible Types with `-xl`

| Pascal Type | FORTRAN Type | Size (bytes) | Alignment (bytes) |
|---|---|---|---|
| real | real | 4 | 4 |
| integer | integer*2 | 2 | 2 |

## *Precautions with Compatible Types*

This section describes the precautions you must take when working with character strings and array indexes.

### *Character Strings*

There are some precautions to take with character strings regarding the null byte, passing by value, and static storage:

- Set the byte count to zero before calling Pascal with a null varying string, because that is what Pascal assumes about such strings.

- Pass a structure consisting of a 4-byte integer and an array of characters from FORTRAN to a Pascal procedure expecting a `var` parameter that is a variable-length string.
- Pass by reference by making the strings `var`, `in`, `out`, or `in out` parameters.
- Set the string to constant because SPARCompiler FORTRAN and Pascal each guarantee the null byte only if the string is a constant. Neither of them rely on the null byte. The null byte is not required by the ISO Pascal Standard.

### *Array Indexes*

The Pascal and FORTRAN array indexes can start at any integer; be sure they match.

## *Incompatibilities*

This section describes the incompatibilities between Pascal and FORTRAN variant records, enumerated types, and set types.

### *Variant Records*

In general, Pascal variant records require adjustment of alignment to match with FORTRAN unions/structures.

### *Enumerated Types*

Pascal enumerated types have no comparable type in FORTRAN.

### *Pascal Set Types*

In Pascal, a set type is implemented as a bit vector, which may be thought of as a FORTRAN 16-bit word. Direct access to individual elements of a set is highly machine-dependent and should be avoided. The implementation may change in a future release.

### *Multidimensional Arrays*

Pascal multidimension arrays are incompatible with FORTRAN multi-dimension arrays. Since Pascal arrays use row-major indexing, and FORTRAN arrays use column-major indexing, an array passed in either direction appears to be transposed.

## *General Parameter Passing in FORTRAN and Pascal*

A few general rules apply to parameter passing:

1. By default, FORTRAN passes all parameters by reference.
2. In FORTRAN, if you want to pass anything by value, then you must explicitly use the nonstandard function `%VAL()`.
3. Pascal passes all parameters by value unless you explicitly state that they are `var`, `out`, or `in out` parameters, in which case they are passed by reference.
4. The routine options `nonpascal`, `extern fortran`, and `external fortran` pass by reference.

## *FORTRAN Calls Pascal*

This section describes how a FORTRAN main program calls a Pascal procedure. It is predominantly made up of examples as follows:

The Pascal procedure, `Samp.p`. Note the procedure definition. The procedure name in the procedure statement is lowercase with trailing underscore (_). This is required to match the conventions of the FORTRAN compiler. Note also that `var` parameters are used to match FORTRAN defaults.

```
procedure samp_(var i: integer; var r: real);

begin
    i := 9;
    r := 9.9
end; { samp_ }
```

The FORTRAN main program, `Sampmain.f`. Note the procedure declaration and call. Also note that FORTRAN converts to lowercase by default and you do not explicitly give the underscore (_).

```
integer i
        double precision  d

        call Samp ( i, d )
        write( *, '(I2, F4.1)')  i, d
        stop
        end
```

The commands to compile and execute `Samp.p` and `Sampmain.f`:

```
hostname% pc -c Samp.p
hostname% f77 Samp.o Sampmain.f -lpfc -lpc
Sampmain.f:
 MAIN:
hostname% a.out
 9 9.9
hostname%
```

## *Variable Parameters*

Pascal passes all `var` parameters by reference, FORTRAN's default.

### *Simple Types without* `-xl`

With `var` parameters, simple types match as in the following example:

The Pascal procedure, `SimVar.p`:

```
procedure simvar_(var t, f: boolean; var c: char;
                  var i: integer; var r: real;
                  var si: integer16; var sr: shortreal);

begin
    t := true;
    f := false;
    c := 'z';
    i := 9;
    r := 9.9;
    si := 9;
    sr := 9.9
end; { simvar_ }
```

The FORTRAN main program, `SimVarmain.f`:

```
      logical*1         t, f
      character         c
      integer*4         i
      double precision  d
      integer*2         si
      real              sr

      call SimVar ( t, f, c, i, d, si, sr )

      write(*, "(L2,L2,A2,I2,F4.1,I2,F4.1)")
      &                  t, f, c, i, d,   si,sr
      stop
      end
```

The commands to compile and execute `SimVar.p` and `SimVarmain.f`:

```
hostname% pc -c SimVar.p
hostname% f77 SimVar.o SimVarmain.f -lpfc -lpc
SimVarmain.f:
 MAIN:
hostname% a.out
 T F z 9 9.9 9 9.9
hostname%
```

### *Simple Types with* `-xl`

With `-xl`, the Pascal `real` must be paired with a FORTRAN `real`, and the Pascal `integer` must be paired with a FORTRAN `integer*2`.

### *Strings of Characters*

The FORTRAN counterpart to the Pascal `alfa` and `string` types is a character string, and the FORTRAN counterpart to the Pascal `varying` is a structure. By default FORTRAN passes all by reference:

The Pascal procedure, `StrVar.p`:

```
type
    varstr = varying [25] of char;

procedure strvar_(var a: alfa; var s: string;
                  var v: varstr);

begin
    a := 'abcdefghij';
    s := 'abcdefghijklmnopqrtstuvwxyz';
    v := 'oyvay'
end; { strvar_ }
```

The FORTRAN main program, `StrVarmain.f`:

```
        structure /VarLenStr/
           integer   nbytes
           character a*25
        end structure
        record /VarLenStr/ vls
        character s10*10, s80*80, s25*25
        vls.nbytes = 0
        Call StrVar( s10, s80, vls )
        s25(1:5) = vls.a(1:vls.nbytes)
        write (*, 1) s10, s80, s25
 1      format("s10='", A, "'",
&            / "s80='", A, "'",
&            / "s25='", A, "'"  )
        end
```

The commands to compile and execute `StrVar.p` and `StrVarmain.f`:

```
hostname% pc -c StrVar.p
hostname% f77 StrVar.o StrVarmain.f -lpfc -lpc
StrVarmain.f:
    MAIN:
hostname% a.out
s10='abcdefghij'
s80='abcdefghijklmnopqrtstuvwxyz
s25='oyvay'
hostname%
```

## *Fixed Arrays*

For a fixed array parameter, pass the same type and size by reference, as shown in the following example:

The Pascal procedure, `FixVec.p`:

```
type
    VecTyp = array [0..8] of integer;

procedure fixvec_(var V: VecTyp; var Total: integer);

var
    i: integer;

begin
    Total := 0;
    for i := 0 to 8 do
      Total := Total + V[i]
end; { fixvec_ }
```

The FORTRAN main program, `FixVecmain.f`:

```
        integer Sum
        integer a(9)
        data    a / 1,2,3,4,5,6,7,8,9 /
        call FixVec ( a, Sum )
        write( *, "( I3 )")  Sum
        stop
        end
```

The commands to compile and execute `FixVec.p` and `FixVecmain.f`:

```
hostname% pc -c FixVec.p
hostname% f77 FixVec.o FixVecmain.f -lpfc -lpc
hostname% a.out
FixVecmain.f:
 MAIN:
    45
hostname%
```

## The univ *Arrays*

You can pass any size array to a Pascal procedure expecting a univ array, although there is no special gain in doing it, since there is no type/size checking for separate compilations. However, if you want to use an existing Pascal procedure that has a univ array, you can do so. All univ arrays that are in, out, in out, or var parameters pass by reference.

The Pascal procedure, UniVec.p, which defines a 10-element array.

```
type
    VecTyp = array [0..9] of integer;

procedure univec_(in V:univ VecTyp; var Last: integer;
                  var Total: integer);

var
    i: integer;

begin
    Total := 0;
    for i := 0 to Last do
      Total := Total + V[i]
end; { univec_ }
```

The FORTRAN main program, UniVecmain.f, which passes a 3-element array to the Pascal procedure written to do a 10-element array.

```
        integer  Sum
        integer a(0:2)
        data    a / 7, 8, 9 /
        call UniVec ( a, 2, Sum )
        write( *, "( I3 )")  Sum
        stop
        end
```

The commands to compile and execute UniVec.p and UniVecmain.f:

```
hostname%  pc -c UniVec.p
hostname% f77 UniVec.o UniVecmain.f -lpfc -lpc
UniVecmain.f:
    MAIN:
hostname% a.out
 24
hostname%
```

## *Conformant Arrays*

For conformant arrays, with single-dimension array, pass upper and lower bounds, placed after the declared parameter list:

```
function ip(var x:array[lb..ub:integer] of real):real;
     ...

double precision v1(10)
double precision z
z = ip ( v1, %VAL(0), %VAL(9) )
...
```

Pascal passes the bounds by value, so FORTRAN must pass them by value too.

One bounds pair may apply to several arrays if they are declared in the same parameter group:

```
function ip(var x,y:array[lb..ub:integer] of real):real;
    ...

double precision v1(10), v2(10)
double precision z
z = ip ( v1, v2, %VAL(0), %VAL(9) )
    ...
```

Examples of single-dimension array and array of character conformant arrays follow. Conformant arrays are included here only because they are a relatively standard feature; there are usually more efficient and simpler ways to do it.

### *Example 1: Single-Dimension Array*

The Pascal procedure, `IntCA.p`. Pascal passes the bounds by value.

```
procedure intca_(var a: array [lb..ub: integer] of integer);

begin
    a[1] := 1;
    a[2] := 2
end; { intca_ }
```

The FORTRAN main program, `IntCAmain.f`:

```
        integer k
        integer s(0:2)
        data    s  / 0, 0, 0 /
        call IntCA ( s, %VAL(0), %VAL(2) )
        do k = 0, 2
        write( *, "(I1)" )  s(k)
        end do
        stop
        end
```

The commands to compile and execute `IntCA.p` and `IntCAmain.f`:

```
hostname% pc -c IntCA.p
hostname% f77 IntCA.o IntCAmain.f -lpfc -lpc
IntCAmain.f:
 MAIN:
hostname% a.out
0
1
2
hostname%
```

### *Example 2: Array of Characters*

The Pascal procedure, `ChrCA.p`. Pascal passes the bounds by value.

```
procedure chrca_(var a: array [lb..ub: integer] of char);

begin
    a[0] := 'T';
    a[13] := 'o'
end; { chrca_ }
```

The FORTRAN main program, `ChrCAmain.f`:

```
      character s*16
      data  s / "this is a string" /
      call ChrCA( s, %VAL(0), %VAL(15) )
      write( *, "(A)" )  s
      stop
      end
```

The commands to compile and execute `ChrCA.p` and `CharCAmain.f`:

```
hostname% pc -c ChrCA.p
hostname% f77 ChrCA.o ChrCAmain.f -lpfc -lpc
ChrCAmain.f:
   MAIN:
hostname% a.out
This is a string
hostname%
```

## *Records/Structures*

In most cases, a Pascal record describes the same objects as its FORTRAN structure equivalent, provided that the components have compatible types and are declared in the same order. The compatibility of the types depends mostly on size and alignment. For more information, see "Compatibility of Types for FORTRAN and Pascal" on page 126.

A Pascal record of an integer and a character string matches a FORTRAN structure of the same things as follows:

The Pascal procedure, `StruChr.p`:

```
type
    lenstr =
      record
          nbytes: integer;
          chrstr: array [0..25] of char
      end;

procedure struchr_(var v: lenstr);

begin
    v.chrstr := 'oyvay';
    v.nbytes := 5
end; { struchr_ }
```

The FORTRAN main program, `StruChrmain.f`:

```
       structure /VarLenStr/
          integer   nbytes
          character a*25
       end structure
       record /VarLenStr/ vls
       character s25*25
       vls.nbytes = 0
       Call StruChr( vls )
       s25(1:5) = vls.a(1:vls.nbytes)
       write ( *, 1 )  s25
 1     format("s25='", A, "'" )
       stop
       end
```

The commands to compile and execute `Struchr.p` and `StruChrmain.f`:

```
hostname% pc -c StruChr.p
hostname% f77 StruChr.o StruChrmain.f -lpfc -lpc
StruChrmain.f:
   MAIN:
hostname% a.out
s25='oyvay'
hostname%
```

## *Variant Records*

FORTRAN equivalents of variant records can sometimes be constructed, although there is some variation with architecture, and sometimes a need to adjust alignment.

The Pascal procedure, `VarRec.p`:

```
type vr = record
    case tag: char of
        'a': ( ch1, ch2: char ) ;
        'b': ( flag: boolean ) ;
        'K': ( ALIGN: integer ) ;
    end ;

procedure varrec_ ( var Rec: vr ) ;

begin
    if ( Rec.ch1 = 'a' )
        then Rec.ch2 := 'Z'
end; { VarRec.p }
```

The FORTRAN main program, `VarRecmain.f`. The variable `ALIGN` is `integer*2` and is needed to match the Pascal variant record layout.

```
      structure /a_var/
              character ch1, ch2
      end structure
      structure /b_var/
              character flag
      end structure
      structure /c_var/
              integer*2 ALIGN
      end structure
      structure /var_part/
              union
                 map
                      record /a_var/ a_rec
                 end map
                 map
                      record /b_var/ b_rec
                 end map
                 map
                      record /c_var/ c_rec
                 end map
              end union
      end structure
      structure /vrnt/
              character tag
              record /var_part/ var_rec
      end structure
      record /vrnt/ VRec
      VRec.var_rec.a_rec.ch1 = 'a'
      VRec.var_rec.a_rec.ch2 = 'b'
      call varrec ( VRec )
      write ( *, * )  VRec.var_rec.a_rec.ch2
      stop
      end
```

The commands to compile and execute `VarRec.p` and `VarRecmain.f` without **`-xl`**:

```
hostname% pc -c VarRec.p
hostname% f77 VarRec.o VarRecmain.f
VarRecmain.f:
   MAIN:
hostname% a.out
b
hostname%
```

## *Pascal Set Type*

The Pascal set type is incompatible with FORTRAN.

## *Pascal* `intset` *Type*

The Pascal `intset` type is predefined as `set of [0..127].` A variable of this type takes a minimum of 16 bytes of storage.

The Pascal procedure, `IntSetVar.p`, which has an `intset` of the elements [1, 3, 7, 8]:

```
procedure intsetvar_(var s: intset);

begin
    s := [1, 3, 7, 8]
end; { intsetvar_ }
```

The FORTRAN main program, `IntSetVarmain.f`:

```
       integer*2 s(8)
       pointer ( ps, s )
       ps = malloc(16)
       call IntSetVar ( s )
       do  i = 5, 8
          write( *, 1 )  s(i), i
       end do
 1     format(o3,1x, 'octal   (word', i2, ')')
       write( *, "('110 001 010 (binary, word 8)')")
       write( *, "('876 543 210 (bit nos, word 8)')")
       stop
```

The commands to compile and execute `IntSetVar.p` and `IntSetVarmain.f`. The output of this example depends on the architecture of your machine. This example was run on a Sun-4™ workstation.

```
hostname% pc -c IntSetVar.p
hostname% f77 IntSetVar.o IntSetVarmain.f -lpfc -lpc
IntSetVarmain.f:
  MAIN:
hostname$ a.out
    0 octal          (word 5)
    0 octal          (word 6)
    0 octal          (word 7)
612 octal            (word 8)
110 001 010          (binary, word 8)
876 543 210          (bit nos, word 8)
hostname$
```

## *Value Parameters*

In general, Pascal passes *value* parameters on the stack.

### *Simple Types Without* `-xl`

Without `-xl`, simple types match as in the following example:

The Pascal procedure, `SimVal.p`. `t`, `f`, `c`, `i`, `r`, and `s` are value parameters.

```
procedure simval_(t, f: boolean; c: char; i: integer;
                  r: real; s: integer16; var reply: integer);

begin
    reply := 0;
    { If nth arg is ok, set nth octal digit to one. }
    if t then
      reply := reply + 1;
    if not f then
      reply := reply + 8;
    if c = 'z' then
      reply := reply + 64;
    if i = 9 then
      reply := reply + 512;
    if r = 9.9 then
      reply := reply + 4096;
    if s = 9 then
      reply := reply + 32768
end; { simval_ }
```

The FORTRAN main program, `SimValmain.f`:

```
        logical*1        t, f
        character        c
        integer*4        i
        double precision d
        integer*2        s
        integer*4        args
        data t / .true. /,  f / .false. /,  c / 'z' /
&            i / 9 /,       d / 9.9 /,      s / 9 /

        call SimVal( %VAL(t), %VAL(f), %VAL(c),
&                    %VAL(i), %VAL(d), %VAL(s), args )
        write( *, 1 )  args
 1      format('args=', o6, '(If nth digit=1, arg n OK)')
        stop
        end
```

The commands to compile and execute `SimVal.p` and `SimValmain.f`:

```
hostname% pc -c SimVal.p
hostname% f77 SimVal.o SimValmain.f -lpfc -lpc
SimValmain.f:
   MAIN:
hostname% a.out
args=111111(If nth digit=1, arg n OK)
hostname%
```

### *Simple Types with* `-xl`

With `-xl`, match Pascal `real` with FORTRAN `real` and Pascal `integer` with FORTRAN `integer * 2`.

You can pass by value using the `%VAL()` feature of Sun FORTRAN.

### *Type* `shortreal`

There is no problem with passing `shortreal` value parameters between Pascal and FORTRAN unlike C. They can be passed exactly as in the previous example, with the Pascal `shortreal` type matching the FORTRAN `real` type.

### *Arrays*

Since FORTRAN cannot pass arrays by *value*, it cannot pass strings of characters, fixed arrays, or `univ` arrays by value.

### *Conformant Arrays*

Although Pascal generally passes all *value* parameters on the stack, the exception is value conformant array parameters, which are handled by creating a copy in the caller environment and passing a pointer to the copy. In addition, the bounds of the array must be passed (see “Conformant Arrays” on page 135).

This example is the same as the one in the earlier section, except that the `var` prefix is deleted.

Pascal procedure, `ChrCAx.p`:

```
procedure chrca_ ( a: array [lb..ub:integer] of char) ;

begin
        a[0] := 'T' ;
        a[13] := 'o' ;
end; { chrca_ }
```

The FORTRAN main program, `ChrCAmain.f`:

```
        character s*16
        data  s / "this is a string" /
        call ChrCA( s, %VAL(0), %VAL(15) )
        write( *, "(A)" )  s
        stop
        end
```

The commands to compile and execute `ChrCAx.p` and `ChrCAmain.f`:

```
hostname% pc -c ChrCAx.p
hostname% f77 ChrCAx.o ChrCAmain.f -lpfc -lpc
ChrCAmain.f:
   MAIN:
hostname% a.out
This is a string
hostname%
```

## *Pointers*

Pointers are easy to pass, as shown in the following example:

The Pascal procedure, `PassPtr.p`. In the Pascal procedure statement, the name must be all lowercase, with trailing underscore (_).

```
type
    PtrInt  = ^integer ;
    PtrReal = ^real ;
procedure passptr_ ( var iPtr: PtrInt ;
                     var dPtr: PtrReal ) ;
begin
    iPtr^ := 9 ;
    dPtr^ := 9.9 ;
end ;
```

The FORTRAN main program, `PassPtrmain.f`. In the FORTRAN main program, uppercase or lowercase of the name is ignored (all converted to lower).

```
      program PassPtrmain
      integer          i
      double precision d
      integer          iptr, dptr
      pointer ( iPtr, i ), ( dPtr, d )
      iPtr = malloc( 4 )
      dPtr = malloc( 8 )
      i = 0
      d = 0.0
      call PassPtr ( iPtr, dPtr )
      write( *, "(i2, f4.1)" )  i, d
      stop
      end
```

The commands to compile and execute `PastPtr.p` and `PassPtrmain.f`:

```
hostname% pc -c PassPtr.p
hostname% f77 PassPtr.o PassPtrmain.f -lpfc -lpc
PassPtrmain.f:
   MAIN passptrmain:
hostname% a.out
9 9.9
hostname%
```

## Function Return Values

Function return values match types the same as with parameters, and they pass in much the same way. See “FORTRAN Calls Pascal” on page 129.

### Simple Types

The simple types pass in a straightforward way as follows:

The Pascal function, `RetReal.p`:

```
function retreal_(var x: real): real;

begin
    retreal_ := x + 1
end; { retreal_ }
```

The FORTRAN main program, `RetRealmain.f`:

```
       double precision r, s, RetReal
       r = 2.0
       s = RetReal( r )
       write( *, "(2f4.1)") r, s
       stop
       end
```

The commands to compile and execute `RetReal.p` and `RetRealmain.f` without `–xl`:

```
hostname% pc -c RetReal.p
hostname% f77 RetReal.o RetRealmain.f -lpfc -lpc
RetRealmain.f:
   MAIN:
hostname% a.out
 2.0 3.0
hostname%
```

### *Type* `shortreal`

There is no problem with returning a `shortreal` function value between Pascal and FORTRAN. It can be passed exactly as in the previous example, with the Pascal `shortreal` type matching the FORTRAN `real` type (without `-xl`).

## *Pascal Calls FORTRAN*

This section parallels “FORTRAN Calls Pascal and the comments and restrictions given in that section apply here, also.

### *Variable Parameters*

#### *Simple Types*

Pascal passes all `var` parameters by reference, the FORTRAN default.

Simple types pass in a straightforward manner as follows:

The FORTRAN subroutine, `SimVar.f`:

```
      subroutine SimVar ( t, f, c, i, d, si, sr )
      logical*1        t, f
      character        c
      integer          i
      double precision d
      integer*2        si
      real             sr
      t = .true.
      f = .false.
      c  = 'z'
      i  = 9
      d  = 9.9
      si = 9
      sr = 9.9
      return
      end
```

The Pascal main program, `SimVarmain.p`:

```
program SimVarmain(output);

var
    t, f: boolean;
    c: char;
    i: integer;
    r: real;
    si: integer16;
    sr: shortreal;

procedure simvar(var t, f: boolean; var c: char;
                 var i: integer; var r: real;
                 var si: integer16; var sr: shortreal);
                 external fortran;

begin
    simvar(t, f, c, i, r, si, sr);
    writeln(t, f: 6, c: 2, i: 2, r: 4: 1, si: 2, sr: 4: 1)
end. { SimVarmain }
```

The commands to compile and execute `SimVar.p` and `SimVarmain.p`:

```
hostname% f77 -c SimVar.f
SimVar.f:
     simvar:
hostname% pc SimVar.o SimVarmain.p -lpfc -lF77
hostname% a.out
true false z 9 9.9 9 9.9
hostname%
```

## *Strings of Characters*

The `alfa` and `string` types pass simply; varying strings are a little tricky. All pass by reference.

The FORTRAN subroutine, `StrVar.f`:

```
        subroutine StrVar ( s10, s80, vls )
        character   s10*10, s80*80
        structure /VarLenStr/
           integer  nbytes
           character a*25
        end structure
        record /VarLenStr/ vls
        character ax*10, sx*80, vx*5
        data ax  / "abcdefghij" /,
&          sx / "abcdefghijklmnopqrstuvwxyz" /,
&          vx / "oyvay" /
        s10(1:10) = ax(1:10)
        s80(1:80) = sx(1:80)
        vls.a(1:5) = vx(1:5)
        vls.nbytes = 5
        return
        end
```

The Pascal main program, `StrVarmain.p`:

```
program StrVarmain(output);

type
    varstr = varying [25] of char;

var
    a: alfa;
    s: string;
    v: varstr;

procedure strvar(var xa: alfa; var xs: string;
                var xv: varstr); external fortran;

begin
    strvar(a, s, v);
    writeln(a);
    writeln(s);
    writeln(v);
    writeln('length(v)= ', length(v): 2)
end. { StrVarmain }
```

The commands to compile and execute `StrVar.f` and `StrVarmain.p`:

```
hostname% f77 -c StrVar.f
StrVar.f:
       strvar:
hostname% pc StrVar.o StrVarmain.p -lpfc -lF77
hostname% a.out
abcdefghij
abcdefghijklmnopqrstuvwxyz
oyvay
length(v)= 5
hostname%
```

## *Fixed Arrays*

For a fixed-array parameter, pass the same type and size by reference:

The FORTRAN subroutine, `FixVec.f`:

```
        subroutine FixVec ( V, Sum )
        integer Sum
        integer V(0:8)
        integer i
        Sum = 0
        do  2 i = 0, 8
 2      Sum = Sum + V(i)
        return
        end
```

The Pascal main program, `FixVecmain.p`:

```
program FixVecmain(output);

type
    VecTyp = array [0..8] of integer;

var
    V: VecTyp := [1, 2, 3, 4, 5, 6, 7, 8, 9];
    Sum: integer;

procedure fixvec(var XV: VecTyp; var XSum: integer);
    external fortran;

begin
    fixvec(V, Sum);
    writeln(Sum: 4)
end. { FixVecmain }
```

The commands to compile and execute `FixVec.f` and `FixVecmain.p`:

```
hostname% f77 -c FixVec.f
FixVec.f:
     fixvec:
hostname% pc FixVec.o FixVecmain.p -lpfc -lF77
hostname% a.out
   45
hostname%
```

## The `univ` *Arrays*

*The* `univ` arrays that are `in`, `out`, `in out`, or `var` parameters pass by reference.

The FORTRAN subroutine, `UniVec.f`:

```
        subroutine UniVec ( V, Last, Sum )
        integer V(0:2), Last, Sum, i
        Sum = 0
        do i = 0, Last
                Sum = Sum + V(i)
        end do
        return
        end
```

The Pascal main program, `UniVecmain.p`:

```
program UniVec;

type
    VecTyp = array [0..9] of integer;

procedure univec(var V:univ VecTyp; in Last: integer;
                 var Sum: integer); external fortran;

var
    Sum: integer;
    V: array [0..2] of integer;

begin
    V[0] := 7;
    V[1] := 8;
    V[2] := 9;
    univec(V, 2, Sum);
    writeln(Sum)
end. { UniVec }
```

The commands to compile and execute `UniVec.f` and `UniVecmain.p`:

```
hostname% f77 -c UniVec.f
UniVec.f:
     univec:
hostname% pc UniVec.o UniVecmain.p -lpfc -lF77
hostname% a.out
     24
hostname%
```

## *Conformant Arrays*

Pascal conformant array parameters are not compatible if Pascal calls FORTRAN.

## *Records/Structures*

Records / structures pass as follows:

The FORTRAN subroutine, `StruChr.f`:

```
      subroutine StruChr ( vls )
      structure /VarLenStr/
         integer   nbytes
         character a*25
      end structure
      record /VarLenStr/ vls
      vls.a(1:5) = 'oyvay'
      vls.nbytes = 5
      return
      end
```

The Pascal main program, `StruChrmain.p`:

```
program StruChrmain;

type
    lenstr =
      record
          nbytes: integer;
          chrstr: array [0..25] of char
      end;

var
    v: lenstr;

procedure struchr(var v: lenstr);
    external fortran;

begin
    struchr(v);
    writeln('v.chrstr = "', v.chrstr, '"');
    writeln('v.nbytes =', v.nbytes: 2)
end. { StruChrmain }
```

The commands to compile and execute `StruChr.f` and `StruChrmain.p`:

```
hostname% f77 -c StruChr.f
StruChr.f:
     struchr:
hostname% pc StruChr.o StruChrmain.p -lpfc -lF77
hostname% a.out
v.chrstr = "oyvay"
v.nbytes = 5
hostname%
```

## *Variant Records*

FORTRAN equivalents of variant records can sometimes be constructed, although there is some variation with architecture, and sometimes a need to adjust alignment. Chapter 7, “The C–Pascal Interface has an example that matches the following example.

The FORTRAN subroutine, `VarRec.f`. The variable `ALIGN` is `integer*2` and is needed to match the Pascal variant record layout.

```
        subroutine VarRec ( VRec )
        structure /a_var/
                character ch1, ch2
        end structure
        structure /b_var/
                character flag
        end structure
        structure /c_var/
                integer*2 ALIGN
        end structure
        structure /var_part/
                union
                   map
                        record /a_var/ a_rec
                   end map
                   map
                        record /b_var/ b_rec
                   end map
                   map
                        record /c_var/ c_rec
                   end map
                end union
        end structure
        structure /vrnt/
                character tag
                record /var_part/ var_rec
        end structure
        record /vrnt/ VRec
        if ( VRec.var_rec.a_rec.ch1 .eq. 'a' )
&               VRec.var_rec.a_rec.ch2 = 'Z'
        return
        end
```

The Pascal main program, `VarRecmain.p`:

```
program VarRecmain;

type
    vr =
      record
          case tag: char of
              'a': ( ch1, ch2: char );
              'b': ( flag: boolean );
              'K': ( ALIGN: integer )
      end;

var
    Rec: vr;

procedure varrec(var d: vr); external fortran;

begin
    Rec.tag := 'a';
    Rec.ch1 := 'a';
    Rec.ch2 := 'b';
    varrec(Rec);
    writeln(Rec.ch2)
end. { VarRecmain }
```

The commands to compile and execute `VarRec.f` and `VarRecmain.p` without **`-xl:`**

```
hostname%
hostname% f77 -c VarRec.f
VarRec.f:
    varrec:
hostname% pc VarRec.o VarRecmain.p -lpfc -lF77
hostname% a.out
b
hostname%
```

## *Value Parameters*

With `external fortran` on the `procedure` statement, Pascal passes value parameters as FORTRAN expects them.

## *Simple Types*

With `external fortran`, the procedure name in the procedure statement and in the call must be all lowercase, but no "_".

The FORTRAN subroutine, `SimVal.f`:

```
      subroutine  SimVal( t, f, c, i, d, s, reply )
      logical*1        t, f
      character        c
      integer*4        i
      double precision d
      integer*2        s
      integer*4        reply
      reply = 0
      if ( t          ) reply = reply + 1
      if ( .not. f    ) reply = reply + 8
      if ( c .eq. 'z' ) reply = reply + 64
      if ( i .eq. 9   ) reply = reply + 512
      if ( d .eq. 9.9 ) reply = reply + 4096
      if ( s .eq. 9   ) reply = reply + 32768
      return
      end
```

The Pascal main program, `SimValmain.p`:

```
program SimVal(output);

var
    t: boolean := true;
    f: boolean := false;
    c: char := 'z';
    i: integer := 9;
    r: real := 9.9;
    s: integer16 := 9;
    args: integer;

procedure simval(t, f: boolean; c: char; i: integer;
                 r: real; s: integer16; var reply: integer);
                 external fortran;

begin
    simval(t, f, c, i, r, s, args);
    writeln('args=', args: 6 oct, ' (If nth digit=1, arg n OK.)')
end. { SimVal }
```

The commands to compile and execute `SimVal.f` and `SimValmain.p`:

```
hostname% f77 -c SimVal.f
SimVal.f:
     simval:
hostname% pc SimVal.o SimValmain.p -lpfc -lF77
hostname% a.out
args=111111 (If nth digit=1, arg n OK.)
hostname%
```

## *Pointers*

Pointers are easy to pass, as shown in the following example:

The FORTRAN subroutine, `PassPtr.f`. In the FORTRAN subroutine, upper/lowercase of the name is ignored (all converted to lower).

```
    subroutine PassPtr ( iPtr, dPtr )
      integer          i
      double precision d
      pointer ( iPtr, i ), ( dPtr, d )
      i  = 9
      d  = 9.9
      return
      end
```

The Pascal main program, `PassPtrmain.p`. In the Pascal program, where it calls the FORTRAN subroutine, the name must be in lowercase.

```
program PassPtrmain;

type
    PtrInt = ^ integer;
    PtrReal = ^ real;

var
    i: integer := 0;
    r: real := 0.0;
    iP: PtrInt;
    rP: PtrReal;

procedure passptr(var xiP: PtrInt; var xrP: PtrReal);
    external fortran;

begin
    iP := addr(i);
    rP := addr(r);
    passptr(iP, rP);
    writeln(i: 2, r: 4: 1)
end. { PassPtrmain }
```

The commands to compile and execute `PassPtr.f` and `PassPtrmain.p`:

```
hostname% f77 -c PassPtr.f
PassPtr.f:
        passptr:
hostname% pc PassPtr.o PassPtrmain.p -lpfc -lF77
hostname% a.out
 9 9.9
hostname%
```

## *Function Return Values*

Function return values match types the same as with parameters, and they pass in much the same way.

### *Simple Types*

The simple types pass in a straightforward way as follows:

The FORTRAN function, `RetReal.f`:

```
double precision function retreal ( x )
 retreal = x + 1.0
 return
 end
```

The Pascal main program, `RetRealmain.p`:

```
program retrealmain;

var
    r, s: real;

function retreal(x: real): real; external fortran;

begin
    r := 2.0;
    s := retreal(r);
    writeln(r: 4: 1, s: 4: 1)
end. { retrealmain }
```

The commands to compile and execute `RetReal.f` and `RetRealmain.p`:

```
hostname% f77 -c RetReal.f
RetReal.f
          retreal:
hostname% pc RetReal.o RetRealmain.p -lpfc -lF77
hostname% a.out
 2.0 3.0
hostname%
```

### *Type* `shortreal`

There is no problem with returning a `shortreal` function value between Pascal and FORTRAN. It can be passed exactly as in the previous example, with the Pascal `shortreal` type matching the FORTRAN `real` type (without `-xl`).

## *Routines as Parameters*

If the passed procedure is a top-level procedure, it is simply as follows:

The FORTRAN subroutine, `PassProc.f`:

```
    subroutine PassProc ( r, s, prcdr )
      real r, s
      external prcdr
      call prcdr ( r, s )
      return
    end
```

The Pascal main program, `PassProcmain.p`:

```
program PassProcmain;

var
    a, b: real;

procedure passproc(var u: real; var v: real;
    procedure p(var r: real; var s: real));
    external fortran;

procedure AddOne(var x: real; var y: real);

begin
    y := x + 1
end; { AddOne }

begin
    a := 8.0;
    b := 0.0;
    passproc(a, b, AddOne);
    writeln(a: 4: 1, b: 4: 1)
end. { PassProcmain }
```

The commands to compile and execute `PassProc.f` and `PassProcmain.p`:

```
hostname% f77 -c PassProc.f
PassProc.f
       passproc
hostname% pc PassProc.o PassProcmain.p -lpfc -lF77
hostname% a.out
 8.0 9.0
hostname%
```

If the procedure is *not* a top-level procedure, then you do not deal with how to pass it, because that requires allowing for the static link. A procedure or function passed as an argument is associated with a static link to its lexical parent's activation record.

When an outer block procedure or function is passed as an argument, Pascal passes a null pointer in the position normally occupied by the static link of the passed routine. So that procedures and functions can be passed to other languages as arguments, the static links for all procedure or function arguments are placed after the end of the conformant array bounds pairs (if any).

Routines in other languages may be passed to Pascal; a dummy argument must be passed in the position normally occupied by the static link of the passed routine. If the passed routine is not a Pascal routine, the argument is used only as a place holder.

# 8

# *Error Diagnostics* 9

This chapter discusses the error diagnostics of Pascal. Appendix B, “Compiler Error Messages lists all error messages the compiler produces.

## *Compiler Syntax Errors*

This section describes some common syntax errors in Pascal programs and shows how the compiler handles them.

### *Illegal Characters*

Characters such as ‘‘@’’ are not part of Pascal. If they are found in the source program and are not part of a string constant, a character constant, or a comment, they are considered to be illegal characters. This can happen if you leave off a closing string quotation mark (‘).

Most nonprinting characters in your input are also illegal, except in character constants and character strings. Except for the tab and formfeed characters, which are used to format the program, nonprinting characters in the input file print as the character ‘‘?’’ in your listing.

## *String Errors*

Encountering an end-of-line after an opening string quotation mark (') without first encountering the matching closing quote yields the diagnostic "`Unmatched ' for string`". Also, anything enclosed in double quotes (for example, `"hello"`) is treated as a comment and therefore ignored.

Programs containing "#" characters (other than in column 1 or in arbitrary-based integers) can produce this diagnostic. This is because early implementations of Pascal used "#" as a string delimiter. In this version of Pascal, "#" is used for `#include` and preprocessor directives and must be in column 1.

## *Digits in Real Numbers*

Pascal requires digits in real numbers before the decimal point. Thus, the statements `b := .075;` and `c := 05e-10;` generate diagnostics in Pascal:

```
Mon Dec 10 10:46:44 1993 digerr.p:
           5  b:= .075;
e 18740-------------------^---  Digits required before decimal point
           6  c:= .05e-10
e 18740-------------------^---  Digits required before decimal point
```

These constructs are also illegal as data input to variables in `read` statements whose arguments are variables of type `real`, `single`, `shortreal`, `double`, and `longreal`.

## *Replacements, Insertions, and Deletions*

When Pascal encounters a syntax error in the input text, the compiler invokes an error recovery procedure. This procedure examines the input text immediately after the point of error and uses a set of simple corrections to see whether to allow the analysis to continue. These corrections involve replacing an input token with a different token or inserting a token. Most of these changes do not cause fatal syntax errors.

The exception is the insertion of or replacement with a symbol such as an identifier or a number; in these cases, the recovery makes no attempt to determine which identifier or what number should be inserted. Thus, these are considered fatal syntax errors.

The Pascal program, `synerr.p`, which uses `**` as an exponentiation operator.

```
program synerr_example(output);

var i, j are integer;

begin
    for j :* 1 to 20 begin
        write(j);
        i = 2 ** j;
        writeln(i))
    end
end. { synerr_example }
```

`synerr.p` produces a fatal syntax error when you compile it because Pascal does not have an exponentiation operator.

```
hostname% pc synerr.p
Mon Dec 10 10:56:19 1993 synerr.p:
           3  var i, j are integer;
e 18460----------------^---  Replaced identifier with a ':'
           6      for j :* 1 to 20 begin
E 18490-----------------^---  Expected keyword (null)
E 18460-----------------^---  Replaced ':' with a identifier
e 18480-----------------------------^---  Inserted keyword do
           8          i = 2 ** j;
e 18480---------------^---  Inserted keyword if
E 18480----------------------^---  Inserted identifier
e 18480-------------------------^---  Inserted keyword then
           9          writeln(i))
e 18450-------------------------^---  Deleted ')'
```

## *Undefined or Improper Identifiers*

If an identifier is encountered in the input but is undeclared, the error recovery mechanism replaces it with an identifier of the appropriate class.

Further references to this identifier are summarized at the end of the containing procedure or function or at the end of the program. This is the case if the reference occurred in the main program.

Similarly, if you use an identifier in an inappropriate way (for example, if a `type` identifier is used in an assignment statement), `pc` produces a diagnostic and inserts an identifier of the appropriate class. Further incorrect references to this identifier are flagged only if they involve incorrect use in a different way, and `pc` summarizes all incorrect uses in the same way it summarizes uses of undeclared variables.

## *Expected Symbols, Malformed Constructs*

If none of the corrections mentioned above appears reasonable, the error recovery routine examines the input to the left of the point of error to see if there is only one symbol that can follow this input. If this is the case, the recovery prints a diagnostic which indicates that the given symbol was expected.

In cases where none of these corrections resolve the problems in the input, the recovery may issue a diagnostic that indicates "malformed" input. If necessary, the compiler can then skip forward in the input to a place where analysis can continue. This process may cause some errors in the missed text to be skipped.

The Pascal program, `synerr2.p`. Here `output` is misspelled and a is given a FORTRAN-style variable declaration.

```
program synerr2_example(input,outpu);

integer a(10)

begin
    read(b);
    for c := 1 to 10 do
        a(c) := b * c
end. { synerr2_example }
```

These are the error messages you receive when you compile `synerr2.p`. On line 6, parentheses are used for subscripting (as in FORTRAN) rather than the square brackets that are used in Pascal.

The compiler therefore noted that a was not defined as a procedure (delimited by parentheses in Pascal). As it's not permissible to assign values to procedure calls, the compiler diagnosed a malformed statement at the point of assignment.

```
hostname% pc synerr2.p
Mon Dec 10 11:02:04 1993 synerr2.p:
           3  integer a(10)
e 18480-------^---  Inserted '['
E 18490-----------------^---  Expected identifier
           6      read(b);
E 18420----------------^---  Undefined variable
           7      for c := 1 to 10 do
E 18420---------------^---  Undefined variable
           8          a(c) := b * c
E 18420---------------^---  Undefined procedure
E 15010 line 1 -  File outpu listed in program statement but not
declared
In program synerr2_example:
E 18240 a undefined on line 8
E 18240 b undefined on lines 6 8
E 18240 c undefined on lines 7 8
hostname%
```

## *Expected and Unexpected End-of-file*

If the compiler finds a complete program, but there is more (noncomment) text in the input file, then the compiler indicates that an end-of-file was expected. This situation may occur after a bracketing error, or if too many `end`s are present in the input. The message may appear after the recovery says that it "`Expected '.'`" because a period (.) is the symbol that terminates a program.

If severe errors in the input prohibit further processing, the compiler may produce a diagnostic message followed by `QUIT`. Examples include unterminated comments and lines longer than 1024 characters.

The Pascal program, `mism.p`:

```
program mismatch_example(output);

begin
    writeln('***');
    { The next line is the last line in the file. }
    writeln
```

When you compile `mism.p`, the end of file is reached before an `end` delimiter.

```
hostname% pc mism.p
E 26020-------^---  Malformed declaration
  15130-------^---  Unexpected end-of-file - QUIT
```

## Compiler Semantic Errors

The following sections explain the typical formats and terminology used in Pascal error messages. For more detailed descriptions of diagnostic messages, refer to Cooper's *Standard Pascal User Reference Manual.*

### Format of the Error Diagnostics

In the example program above, the error diagnostics from the Pascal compiler include the line number in the text of the program, as well as the text of the error message. While this number is most often the line where the error occurred, it can refer to the line number containing a bracketing keyword like `end` or `until`. If so, the diagnostic may refer to the previous statement. This occurs because of the method the compiler uses for sampling line numbers. The absence of a trailing ";" in the previous statement causes the line number corresponding to the `end` or `until` to become associated with the statement. As Pascal is a free-format language, the line number associations can only be approximate and may seem arbitrary in some cases.

### Incompatible Types

Since Pascal is a strongly-typed language, many type errors can occur, which are called *type clashes* by the compiler. The types allowed for various operators in the language are summarized in the *Standard Pascal User Reference Manual.* It is important to know that the Pascal compiler, in its diagnostics, distinguishes among the following type classes:

```
array    integer  scalar
boolean  pointer  string
char     real     varying
file     record
```

Thus, if you tried to assign an `integer` value to a `char` variable you would receive a diagnostic as follows:

```
Mon Dec 10 13:16:20 1993 inchar.p:
E 25190 line 6 -  Type clash: integer is incompatible with char
  ... 25560: Type of expression clashed with type of variable in assignment
```

In this case, one error produced a two-line error message. If the same error occurs more than once, the same explanatory diagnostic is given each time.

## *The* `scalar` *Class*

The only class whose meaning is not self-explanatory is `scalar`. It has a precise meaning in the Pascal standard where it refers to `char`, `integer`, and `boolean` types as well as the enumerated types. For the purposes of the Pascal compiler, `scalar` in an error message refers to a user-defined enumerated type, such as `color` in

```
type color = (red, green, blue)
```

For integers, the more precise denotation `integer` is used.

## *Procedure and Function Type Errors*

For built-in procedures and functions, two kinds of errors may occur. If a routine is called with the wrong number of arguments, a message such as the following is displayed:

```
Mon Dec 10 13:21:26 1993 sin.p:
E 10010 line 6 -  Builtin function SIN takes exactly 1 argument
```

If the type of an argument is wrong, you receive a message such as the following:

```
Mon Dec 10 13:31:14 1993 abs.p:
E 10180 line 8 -  Argument to ABS must be of type integer or real, not char
```

# 9

## *Procedures and Functions as Parameters*

In standard Pascal, procedures and functions used as formal parameters can be declared with (nested) parameter lists of their own. In Jensen and Wirth's Pascal there are no nested parameter lists; therefore, no argument checking is possible in calls made to parametric procedures and functions. Pascal requires you to use parametric procedures to conform to the standard, so programs ported from early implementations of Pascal may require modification.

## *Scalar Error Messages*

Error messages stating that scalar (user-defined) types cannot be read from and written to files are often mysterious. In fact, if you define

```
type color = (red, green, blue)
```

standard Pascal does not associate these constants with the strings ''red'', ''green'', and ''blue'' in any way. This Pascal adds an extension that allows enumerated types to be read and written; however, if the program is to be portable, you must write your own routines to perform these functions. Standard Pascal only allows the reading of characters, integers, and real numbers from text files including `input` (not strings or booleans). It is possible to make the following:

```
file of color
```

However, the representation is binary rather than as strings and it is impossible to define `input` as other than a text file.

## *Expression Diagnostics*

The diagnostics for semantically ill-formed expressions are explicit as the following program shows. The following Pascal program, `expr.p`, is admittedly far-fetched, but illustrates that the error messages are clear enough to allow you to easily determine the problem in the expressions.

```
program expr_example(output);

var
    a: set of char;
    b: Boolean;
    c: (red, green, blue);
    p: ^ integer;
    A: alfa;
    B: packed array [1..5] of char;

begin
    b := true;
    c := red;
    new(p);
    a := [];
    A := 'Hello, yellow';
    b := a and b;
    a := a * 3;
    if input < 2 then writeln('boo');
    if p <= 2 then writeln('sure nuff');
    if A = B then writeln('same');
    if c = true then writeln('hue''s and color''s')
end. { expr_example }
```

This program generates the following error messages:

```
hostname% pc expr.p
Mon Dec 10 13:36:51 1993 expr.p:
E 13050 line 16 -  Constant string too long
E 20070 line 17 -  Left operand of and must be Boolean, not set
E 25550 line 18 -  expression has invalid type
E 20030 line 18 -  Cannot mix sets with integers and reals as operands of *
E 20240 line 19 -  files may not participate in comparisons
E 20230 line 20 -  pointers and integers cannot be compared - operator was <=
E 20220 line 21 -  Strings not same length in = comparison
E 20230 line 22 -  scalars and Booleans cannot be compared - operator was =
hostname%
```

# 9

## *Type Equivalence*

The Pascal compiler produces several diagnostics that generate the following message:

```
non-equivalent types
```

In general, Pascal considers types to be the same only if they derive from the same type identifier. Therefore, the following two variables have different types, even though the types look the same and have the same characteristics.

```
x : record
       a: integer;
       b: char;
    end;
y :  record
       a: integer;
       b: char;
    end;
```

The assignment

```
x := y
```

produces the following diagnostic messages:

```
Mon Dec 10 14:22:46 1993 inchar.p:
E 25170 line 12 -  Type clash: non-identical record types
   ... 25560: Type of expression clashed with type of variable in assignment
```

To make the assignment statement work, you need to declare a type and use it to declare the variables.

```
type
    r = record
       a: integer;
       b: char;
    end;
var
    x: r;
    y: r;
```

Alternatively, you could use the declaration:

```
x, y : record
       a: integer;
       b: char;
    end;
```

and the assignment statement will work.

## *Unreachable Statements*

Pascal flags unreachable statements. Such statements usually correspond to errors in the program logic, as shown in the following example:

The Pascal program, `unreached.p`:

```
program unreached_example(output);

label
    1;

begin
    goto 1;
    writeln('Unreachable.');
    1:
    writeln('Reached this.');
end. { unreached_example }
```

`unreached.p` generates these error messages when you compile it :

```
hostname% pc unreached.p
Tue Dec 11 14:21:03 1993 unreached.p:
w 18630 line 8 -  Unreachable statement
hostname%
```

A statement is considered to be reachable if there is a potential path of control, even if it can never be taken. Thus, no diagnostic is produced for the statement

```
if false then
writeln('Impossible!')
```

# 9

### *The* `goto` *Statement*

The compiler detects and complains about `goto` statements that transfer control into structured statements (for example, `for` and `while`). It does not allow such jumps, nor does it allow branching from the `then` part of an `if` statement into the `else` part. Such checks are made only within the body of a single procedure or function.

### *Uninitialized Variables*

The Pascal compiler, `pc`, does not necessarily set variables to an initial value unless you explicitly request that with the `–Z` option. The exception is `static` variables, which are guaranteed to be initialized with zero values.

Because variable use is not tracked across separate compilation units, `pc` does nothing about uninitialized or unused variables in global scope (that is, in the main program). However, `pc` checks variables with local scope (that is, those declared in procedures and functions) to make sure they are initialized before being used. `pc` flags uninitialized variables with a warning message.

### *Unused Variables, Constants, Types, Labels, and Routines*

If you declare a variable, constant, type, procedure, or function in local scope but never use it, the compiler gives you a warning message. The compiler does not do this for items declared in global scope because you may use the items in a separately compiled unit.

If you declare a label but never use it, the compiler gives you a warning. This is true even for a label declared in global scope.

## *Compiler Panics, I/O Errors*

One class of error that rarely occurs, but that causes termination of all processing when it does, is a *panic*.

A panic indicates a compiler-detected internal inconsistency. A typical panic message follows:

```
pc0 internal error line=110 yyline=109
```

If you receive such a message, compilation is quickly and perhaps ungracefully terminated. Save a copy of your program to inspect later, then contact Customer Support. If you were making changes to an existing program when the problem occurred, you may be able to work around the problem by determining which change caused the internal error and making a different change or error correction to your program.

### *Out of Memory*

The only other error that aborts compilation when no source errors are detected is running out of memory. Most tables in the compiler, with the exception of the parse stack, are dynamically allocated, and can grow to take up a good deal of process space. In general, you can get around this problem with large programs by using the separate compilation facility.

If you receive an ‘‘out of space’’ message from the compiler during translation of a large procedure or function, or one containing a large number of string constants, you can either break the procedure or function into smaller pieces or increase the maximum data segment size using the `limit` command of `csh`(1).

## *Runtime Errors*

When an error is detected in a program during runtime, a message describing the error is printed and the program is aborted, producing a core image. Following is a list of runtime errors that Pascal generates.

- <*filename*> : Attempt to read from a file open for writing
- <*filename*> : Attempt to write to a file open for reading
- <*filename*> : Bad data found on enumerated read
- <*filename*> : Bad data found on integer read
- <*filename*> : Bad data found on real read
- <*filename*> : Bad data found on string read

- <*filename*> : Bad data found on varying of `char` read
- <*filename*> : Cannot close file
- <*filename*> : Could not create file
- <*filename*> : Could not open file
- <*filename*> : Could not remove file
- <*filename*> : Could not reset file
- <*filename*> : Could not seek file
- <*filename*> : Could not write to file
- <*filename*> : File name too long (maximum of <*number*> exceeded)
- <*filename*> : File table overflow (maximum of <*number*> exceeded)
- <*filename*> : Line limit exceeded
- <*filename*> : Non-positive format widths are non-standard
- <*filename*> : Overflow on integer read
- <*filename*> : Tried to read past end of file
- <*filename*> : `eoln` is undefined when `eof` is true
- Argument <*number*> is out of the domain of `atan`
- Argument to `argv` of <*number*> is out of range
- Assertion #<*number*> failed: <*assertion message*>
- Cannot close null file
- Cannot compute cos(<*number*>)
- Cannot compute sin(<*number*>)
- Cannot open null file
- Enumerated type value of <*number*> is out of range on output
- Floating point division by zero
- Floating point operand is signaling Not-A-Number
- Floating point overflow
- Floating point underflow

- Illegal argument of <*number*> to trunc
- Inexact floating point result
- Integer divide by zero
- Integer overflow
- Internal error in enumerated type read
- Invalid floating point operand
- Label of <*number*> not found in case
- Negative argument of <*number*> to `sqrt`
- Non-positive argument of <*number*> to `ln`
- Overflow / Subrange / Array Subscript Error
- Pointer value (<*number*>) out of legal range
- Ran out of memory
- Range lower bound of <*number*> out of set bounds
- Range upper bound of <*number*> out of set bounds
- Reference to an inactive file
- Second operand to MOD (<*number*>) must be greater than zero
- Statement count limit of <*number*> exceeded
- Subrange or array subscript is out of range
- Sun FPA not enabled
- Unknown SIGFPE code
- Unordered floating point comparison
- Value of <*number*> out of set bounds
- Varying length string index <*number*> is out of range
- `exp`(<*number*>) yields a result that is out of the range of reals
- i = <*number*>: Bad i to pack(`a,i,z`)
- i = <*number*>: Bad `i` to unpack(`z,a,i`)
- `pack`(`a,i,z`): `z` has <*number*> more elements than `a`

- substring outside of bounds of string
- unpack(`z,a,i`): `z` has <*number*> more elements than `a`

# *Using the X View Toolkit* 10

This chapter introduces the XView programmer's toolkit, a part of the XView Application Programmer's Interface. It assumes you are familiar with XView windows from the user point of view. This chapter introduces XView from a programmer point of view. For complete programmer information, see the *XView Programming Manual*.

## *Overview*

The XView Application Programmer's Interface is an object-oriented, server-based, user-interface toolkit for the X Window System Version 11 (X11). It is designed and documented to help programmers manipulate XView windows and other XView objects. This chapter focuses on using it with Pascal.

### *Tools*

This kit is a collection of functions. The runtime system is based on each application having access to a server-based *Notifier*, which distributes input to the appropriate window, and a *window manager*, which manages overlapping windows. There is also a *Selection Service* for exchanging data between windows (in the same or different processes).

# 10

## Objects

XView is an *object-oriented* system. XView applications create and manipulate XView objects. All such objects are associated with XView packages. Objects in the same package share common properties. The major objects are windows, icons, cursors, menus, scrollbars, and frames. A *frame* contains non-overlapping subwindows within its borders. You manipulate an object by passing a unique identifier or *handle* for that object to procedures associated with that object.

### Definition of Object-Oriented Programming

Traditional programs are made up of procedures. When you need to operate on some data, you pass the data to a procedure. This style of programming can be referred to as *procedure-oriented* programming.

In object-oriented programming, the data are organized into *objects*, which are similar to records in that an object can contain data fields of different types. In addition to data, though, objects also have associated procedures called *methods*. The methods of an object generally perform all operations that can be performed on the data of the object. When you need to operate on the data in an object, you tell the object to do the operation. This is referred to as sending a *message* to the object, and is similar to calling a procedure.

Each object is an *instance* of a given *class*. A class is much like a type in that it defines a kind of object. Creating an instance of a given class is much like declaring a variable of a given type in that creating an instance of a class creates an object that has the properties and characteristics defined for its class.

Classes are different from types in that a class can *inherit* data fields and methods from another class. In fact, you always define a new class by declaring it a *subclass* of some class. The new class is called a *child* or *descendant* class; the old class is called a *superclass*, *parent*, or *ancestor* class. A descendant can itself have descendants. Thus, classes form a tree structure, with the root being the class from which all others are descended. In the XView toolkit, the root is the class Generic Object, which has no data fields and no methods. Each descendant of Generic Object is specialized in some way, perhaps with additional data fields or methods.

What gives object-orientation its power is that the inherited methods of a subclass can be reimplemented so that they take actions suited to the subclass.

For example, consider a system that has three classes: Drawable Object, Window, and Icon. Window and Icon are subclasses of Drawable Object. This creates the hierarchical relationship illustrated in Figure 10-1.

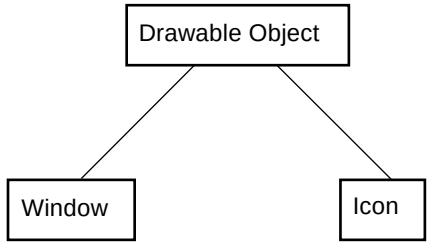


*Figure 10-1* A Sample Class Hierarchy

Suppose that Drawable Object has a method that draws the object. Window and Icon inherit that method but each implements it in a different way. Window defines a Draw method that draws windows; Icon defines a Draw method that draws icons.

When you write your program, you can send a message to an object telling it to draw itself without knowing whether, at runtime, the object is an icon or a window. This works because both Window and Icon are descendants of Drawable Object. At runtime, the object that receives the method will draw itself using its classes implementation of the Draw method.

## *Pascal Interface*

This chapter focuses on manipulating XView windows and objects with Pascal. To write XView applications, you need to use special libraries, modules files, object handles, and standard procedures.

### *Compiling with Libraries*

Most XView procedures you call are in the libraries `pxview`, `xview`, and `X11`. To compile an XView program, link the libraries, in order. For example:

```
hostname% pc my_pascal.p  -lpxview -lxview -lolgx -lX11
```

# 10

## Header Files

The header files define the necessary data types, constants and external procedures necessary to write programs using the XView interface with Pascal.

### Names

The names of the header file are the same as the XView C language header files with the `.h` extension changed to `_p.h`. For example, the Pascal header file corresponding to the XView file `panel.h` is named `panel_p.h`. Other header files are `canvas_p.h, text_p.h,` and so on.

In addition to the header files corresponding to the XView headers, there is an another one, which is:

```
stddefs_p.h
```

This defines some basic types that are used by most of the other Pascal XView header files. This is included by the header files that need it.

### Usage

To use header files with Pascal, put in include lines for any other header files you need.

### Attribute Procedures

Each class of objects has its own set of *attributes*. Each attribute has a predefined, or default, value. For example, for the class of scrollbars, there is a width and a color.

The standard C interface to XView defines two routines `xv_get()` and `xv_set()` which get and set attributes of XView objects. These routines take an arbitrary number and type of parameters and return various types depending on its arguments.

Instead of these routines, the Pascal interface to XView defines a separate routine to get and set each attribute.

*set* — The routine to set an attribute is named `set_`*attrname*

- Each set routine is a procedure.

- Each set routine takes, as its first argument, the object for which the attribute is being set.
- The second argument is the value of the attribute.

*get* — The routine to get the value of an attribute is named `get_`*attrname*

- Each set routine is a function.
- Each get routine takes an XView object as the first argument
- Each get routine returns the value of the attribute requested.

For example,

```
set_WIN_SHOW ( frame, true );

width := get_CANVAS_WIDTH ( canvas );
```

These routines are defined in the header file `attrgetset_p.h`.

## *Attribute Lists*

Some of the XView C routines may optionally take extra arguments that are lists of attributes and values. The extra arguments vary in number and type. You must pass a 0 to the last argument of these routines.

Since Pascal does not support variable length argument lists, the Pascal definition has a single argument.

Instead, special versions of these routines are provided which take as a last argument an argument of type `Attr_avlist`. This type is a pointer to an array of attributes and values. The special routines are

`xv_init_l()`,
`xv_create_l()`,
`xv_find_l()`,
`selection_ask_l()`,
`selection_init_request_l()`.

Example calls

Here, mymenu is an object of type XV_object.

```
mymenu := xv_create (NULL, MENU, 0);
ncols := get_MENU_NCOLS (mymenu);
set_MENU_NITEMS (mymenu, items);
 xv_find_l (mymenu, MENU,
    attr_create_list_2s (MENU_DEFAULT,4));
 xv_destroy (mymenu);
```

The lists for `Attr_avlist` are created by functions that have the names

`attr_create_list_`*n*()

`attr_create_list_`*ns*()

- The *n* indicates the number of arguments the routine accepts.
- The number of arguments can be 1-16.
- The routines ending in `s` return a pointer to a static attribute-value array which is reused with each call to the static list routines.
- The versions without the `s` return a dynamically allocated list which should be passed to `xv_destroy()` when you are finished with it.

## *Handles*

If you create an XView object then `xv_create()` returns a *handle* for that object. You pass this handle to the appropriate procedure for manipulating the object.

## *Data Types*

Each XView object has its own specific data type. The name of an object's data type always starts with a capital letter. For example, the data type for a scrollbar is `Scrollbar`. The standard list of these types is in the header files.

## *Coding Fragment*

This sample is provided to illustrate the style of programming with the XView interface in Pascal. It does three things.

- Creates a vertical scrollbar with a view length of 100 pixels

- Changes the view length to 200 pixels
- Destroys the scrollbar

```
var bar, pi: Scrollbar;

begin
    bar := xv_create ( 0, SCROLLBAR, 0 );
    pi  := xv_create_l ( 0, SCROLLBAR,
     attr_create_list_4s  ( SCROLLBAR_VIEW_LENGTH, 100,
     SCROLLBAR_DIRECTION, SCROLLBAR_VERTICAL));

    set_SCROLLBAR_VIEW_LENGTH( bar, 200 );
    xv_destroy ( bar );
```

Note the following about this example.

- `bar` is declared to be of type `Scrollbar`.
- `xv_create()` and `xv_create_l()` are invoked as *functions.*
- `set SCROLLBAR_VIEW_LENGTH ()` is invoked as a *procedure.*
- `xv_destroy()` is invoked as a *procedure.*
- The zero third argument is set to `xv_create`.

## *C to Pascal*

This section lists a problem that you can face when converting C to Pascal and offers some changes that you can make to help workaround the problem.

**The Problem** — Besides the six standard generic procedures, there are approximately 80 other procedures, plus hundreds of attributes. These are all documented in the manual *XView Programming Manual*. The problem is that all of the coding is in C.

**The Easy Part** — You can use the following items of information as you find them in the manual, with no change.

- The XView procedure names
- The XView object names

- The XView object data types (except `Boolean`, see the following section)

**The Hard Part** — You must make the following changes.

- Any elementary C data type used must be converted to the corresponding Pascal data type.
- Any C procedure that *returns* something must be invoked in Pascal as a function; otherwise it must be invoked as a procedure.
- The XView type `Boolean` must be converted to the Pascal type `boolean`.

Table 10-1 summarizes converting C declarations to Pascal.

*Table 10-1* C Declarations to Pascal Declarations

| C | Pascal |
|---|---|
| `int` | `integer`, subrange, or numeric constant |
| `unsigned` | `unsigned`, subrange, or numeric constant[1] |
| `short` | `integer16`, subrange, or numeric constant |
| `unsigned short` | `unsigned16`, subrange, or nonnegative numeric constant† |
| `char` | `char` (or single-letter string literal for special definition modules) |
| `float` | `shortreal`, `longreal`, or real constant (always passed as `LONGREAL`) |
| `double` | `real` or real constant |
| any pointer type | pointer type |
| any `enum` type | `unsigned` type |

1. Defined in `stddefs_p.h`.

| | |
|---|---|
| any `struct` type | `record` type of corresponding size and layout |
| `char *` | `array of char` or string literal |
| other array types | `array` type of corresponding size |

## *Sample Translation of an XView Function to Pascal*

In the manual *XView Programming Manual*, in the chapter "XView Procedures and Macros," you find this entry:

```
texsw_insert()
    Inserts characters in buf into textsw at the current
    insertion point. The number of characters actually
    inserted is returned. This will equal buf_len unless
    there was a memory allocation failure. If there was
    a failure, it will return 0.

    Textsw_index
    textsw_insert(textsw, buf, buf_len)
       Textsw         textsw;
       char           *buf;
       int            buf_len;
```

Translation to Pascal:

- Leave the object data type `Textsw` as is.
- Since it returns the number of characters inserted, invoke it as an integer function.

```
    ...

var
    textsw: Textsw   ;
    buf: array[0..3] of char ;
    buf_len:  integer;
    N:Textsw_index;

begin
    ...
    N := textsw_insert (textsw, buf, buf_len) ;
    end.
```

## *Sample Program*

The following example, `xview.p`, makes a window.

```
program hello(output);

#include "stddefs_p.h"
#include "attrgetset_p.h"

var
  base_frame :Frame;
  base_panel :Panel;
  message    :Xv_panel_or_item_ptr;
  text       :string;

begin
  text := 'Hello, World!';

  xv_init(0);

  base_frame := xv_create(nil, FRAME, 0);
  base_panel := xv_create(base_frame, PANEL, 0);
  message    := xv_create(panel, PANEL_MESSAGE, 0);
    set_PANEL_LABEL_STRING(message, text);

  window_main_loop(base_frame);

end. {hello}
```

To compile `xview.p` and link in the necessary libraries, use the following command line. Replace *local_library_path* with the path for the Pascal XView libraries on your system.

```
hostname% pc -Ilocal_library_path xview.p -L$OPENWINHOME/lib \
-lpxview -lxview -lolgx -lX11
```

Now run the executable file using the following command line.

```
hostname% a.out
```

Soon after you run the executable file, the window will come up as a single frame, with the string “Hello, World!” in the frame header.

# 10

# *Pascal Preprocessor* A

This appendix describes the preprocessors cpp(1) and cppas.

## cpp

cpp(1) is the C language preprocessor. Pascal runs your source program through cpp(1) when you compile it without the –xl option. For a complete description of cpp(1), see the *SunOS 5.2 Reference Manual.*

## cppas

The cppas preprocessor handles the Pascal conditional variables and compiler directives. You call cppas using the –xl option.

### *Conditional Variables*

A conditional variable is defined when it appears in a %var directive; otherwise, it is undefined. In addition, we predefine:

| | | |
|---|---|---|
| __sun | __SVR4 | sparc |
| __sparc | __SUNPRO_PC=0x302 | unix |
| __unix | sun | |

These variables are not predefined when you use -s0, -s1, -V0, or -V1.

A defined conditional variable is enabled (true) when it appears in either the `%enable` directive or in the `–config` option; otherwise, it is disabled (false).

```
%var one two
%enable two
```

The following section describes `%var` and `%enable`. Programs that contain conditional variables must be compiled with the `–xl` option.

## *Compiler Directives*

A directive indicates some action for the compiler to take. You can use a directive anywhere in your program.

Each directive consists of a percent sign (%) followed by the directive name. Programs that contain compiler directives must be compiled with the `–xl` option.

Table A-1 summarizes the compiler directives. The remaining sections of this appendix contain detailed descriptions and examples of each directive.

*Table A-1* The `cppas` Compiler Directives

| **Compiler Directive** | **Description** |
|---|---|
| %config | Special predefined conditional variable with a value of either true or false. |
| %debug | Tells `pc` to compile this line of code when you use the `-cond` compiler directive. |
| %else | If the *expression* in `%if` *expression* `%then` is false, the compile skips over the `%then` part and executes the `%else` part instead. |
| %elseif | Similar to `%else`. If the *expression* in `%if` *expression* `%then` is false, the compiler skips over the `%then` part and executes the `%elseif` part instead. |
| %elseifdef | If the *expression* in `%ifdef` *expression* `%then` is false, the compiler skips over the `%then` part and executes the `%elseifdef` part instead. |
| %enable | Sets a conditional variable to true. |
| %endif | Indicates the end of an `%if` or `%ifdef` directive. |

*Table A-1* The `cppas` Compiler Directives

| Compiler Directive | Description |
|---|---|
| %error | Prints a string on the standard output and treats it as an error. |
| %exit | Stops processing the current Pascal source file. |
| %if | When the compiler encounters an `%if` *expression* `%then` directive, it evaluates e*xpression*. If *expression* is true, the compiler executes the statements after `%then`. If *expression* is false, the compiler skips over `%then`. |
| %ifdef | Determines whether or not you previously defined a conditional variable in a `%var` directive. |
| %include | Inserts the lines from the specified file into the input stream. |
| %list | Enables a listing of the program. |
| %nolist | Disables the program listing. |
| %slibrary | Inserts the lines from the specified file into the input stream. Same as `%include`. |
| %var | Defines conditional variables. |
| %warning | Prints a warning string on the standard output. |

## *The `%config` Directive*

The `%config` directive is a predefined conditional variable with a value of either true or false.

### *Syntax*

%config

### *Comments*

`%config` is true when you compile your program with the `-config` option; otherwise, `%config` is false.

Use `%config` in an `%if`, `%ifdef`, `%elseif`, or `%elseifdef` directive to catch any undefined values specified with `-config`. Do not define `%config` in the `%var` directive.

### *Example*

The Pascal program, `config.p`, which defines the conditional variables `one` and `two`:

```
program config_example(output);

{ This program demonstrates the use of the
  %config compiler directive. }

var
    a: integer := maxint;
    b: integer := minint;

%var one two

begin
    writeln('Begin program.');
    %if one %then
        writeln('One is defined as ', a:2, '.');
    %elseif two %then
        writeln('Two is defined as ', b:2, '.');
    %elseif %config %then
        writeln('Nothing is defined.');
    %endif
    writeln('End program.')
end. { config_example }
```

The output when you compile `config.p` without the `-config` option:

```
hostname% pc -xl config.p
hostname% a.out
Begin program.
End program.
hostname%
```

The output when you define the variable `one`:

```
hostname% pc -xl -config one config.p
hostname% a.out
Begin program.
One is defined as 32767.
End program.
hostname%
```

The output when you define `two`:

```
hostname% pc -xl -config two config.p
hostname% a.out
Begin program.
Two is defined as -32768.
End program.
hostname%
```

The output when you define `foo`:

```
hostname% pc -xl -config foo config.p
Tue Dec 2 15:22 1993 config.p

Error: –CONFIG command argument foo was never declared.
Compilation failed
hostname%
```

## The %debug *Directive*

The %debug directive tells pc to compile this line of code when you use the –cond compiler directive.

### *Syntax*

%debug;

### *Comments*

The %debug directive works in conjunction with the –cond compiler option. –cond tells pc to compile the lines in your program that begin with %debug. Without –cond, pc treats lines with %debug as comments.

***Example***

The Pascal program, `debug.p`:

```
program debug_example(output);

{ This program demonstrates the use of the
  %debug compiler directive. }

begin
    writeln ('Hello, how are you?');
    %debug;  writeln ('Fine, thank you.');
end. { debug example }
```

The output when you compile `debug.p` without the `-cond` option:

```
hostname% pc -xl debug.p
hostname% a.out
Hello, how are you?
hostname%
```

The output when you use –cond:

```
hostname% pc -xl –cond debug.p
hostname% a.out
Hello, how are you?
Fine, thank you.
hostname%
```

## *The %else Directive*

The `%else` directive provides an alternative action to the `%if` directive.

***Syntax***

`%if` *expression* `%then`
    .
    .
`%else`
    .
    .
`%endif`

***Example***

The Pascal program, `if_then_else.p`:

```
program if_then_else (output);
%var red
begin
%if red %then
  writeln ('It is red.');
%else
  writeln ('It is not red.')
%endif
end.
```

The output when you compile `if_then_else.p` without the `-config`:

```
hostname% pc -xl if_then_else.p
hostname% a.out
It is not red.
hostname
```

The output when you supply `-config` with the argument `red`:

```
hostname% pc -xl -config red if_then_else.p
hostname% a.out
It is red.
hostname%
```

## *The %elseif Directive*

The `%elseif` directive provides an alternative action to the `%if` directive.

***Syntax***

```
%if expression %then
      .
      .
%elseif expression %then
      .
      .
%endif
```

### *Comments*

If the expression in `%if` *expression* `%then` is false, the compiler skips over the `%then` part and executes the `%elseif` part instead. e*xpression* consists of a conditional variable and the optional boolean operators `and`, `or`, and `not`. See the `%else` listing for examples of *expression*.

### *Example*

The Pascal program, `elseif.p`:

```
program elseif_example(output);

{ This program demonstrates the use of the
   %if, %then, and %elseif directives. }

%var blue red

begin
    %if blue %then
        writeln('The color is blue.');
    %elseif red %then
        writeln('The color is red.');
    %endif
end. { elseif_example }
```

The output when you supply `-config` with the argument `blue`:

```
hostname% pc -xl -config blue elseif.p
hostname% a.out
The color is blue.
```

The output when you supply `-config` with the argument `red`:

```
hostname% pc -xl -config red elseif.p
hostname% a.out
The color is red.
```

## *The %elseifdef Directive*

The `%elseifdef` directive provides an alternative action to the `%ifdef` directive.

### *Syntax*

```
%ifdef expression %then
       .
       .
%elseifdef expression %then
       .
       .
%endif
```

### *Comments*

If the expression in `%ifdef` *expression* `%then` is false, the compiler skips over the `%then` part and executes the `%elseifdef` part instead. The e*xpression* consists of a conditional variable and the optional boolean operators `and`, `or`, and `not`. See the `%else` listing for examples of *expression*.

### *Example*

The Pascal program, `ifdef.p`, which first checks if `bird1` has been defined. If not, it defines it with a `%var` directive. If `bird1` has been defined, the program checks whether or not it needs to define `bird2`.

```
program ifdef_example(output);

%include 'bird.h';

begin
    %ifdef not(bird1) %then
          %var bird1
    %elseifdef not(bird2) %then
          %var bird2
    %endif;

    %if bird1 %then
        writeln('Bird one is a ', a, '.');
    %elseif bird2 %then
        writeln('Bird two is a ', b, '.')
    %endif
end.  { ifdef_example }
```

The `include` file, `bird.h`:

```
var
    a: array[1..7] of char := 'penguin';
    b: array[1..6] of char := 'toucan';

%var bird1
```

The output when you enable `bird1` with the `-config` option:

```
hostname% pc -xl -config bird1 ifdef.p
hostname% a.out
Bird two is a penguin.
```

The output when you enable `bird2`:

```
hostname% pc -xl -config bird2 ifdef.p
hostname% a.out
Bird two is a toucan.
```

## *The %enable Directive*

The `%enable` directive sets a conditional variable to true.

### *Syntax*

`%enable` *var1 ..., varN*

### *Comments*

A defined conditional variable is `enable` (true) when it appears in either the `%enable` directive or in the `-config` option. Conditional variables are false by default.

***Example***

The Pascal program, `enable.p`. This example sets the conditional variable `two` to true, which is equivalent to setting the `-config` option to `two` on the command line.

```
program enable_example(output);

{ This program demonstrates the use of
   the %enable compiler directive. }

var
    a: integer;
    b: integer;

%var one, two
%enable two

begin
    %if one %then
        a := maxint;
        writeln('One is defined as ', a:2, '.');
    %endif
    %if two %then
        b := minint;
        writeln('Two is defined as ', b:2, '.');
    %endif
end. { enable_example }
```

The commands to compile and output `enable.p`:

```
hostname% pc -xl enable.p
hostname% a.out
Two is defined as -32768.
hostname%
```

## *The %endif Directive*

The `%endif` directive indicates the end of a `%if` or `%ifdef` directive. See the sections on `%if` and `%ifdef` for more information on this directive.

## *The %error Directive*

The `%error` directive tells the compiler to print a string on the standard output and treat it as an error.

### *Syntax*

`%error '`*string*`'`

### *Comments*

The compiler does not produce an object file.

### *Example*

The Pascal program, `error.p`:

```
program error_example(output);

{ This program demonstrates the use of the
  %error compiler directive. }

%var arch

begin
    %if arch %then
        writeln('This is a SPARC computer.');
    %else
        %error 'Unknown architecture.'
    %endif
end. { error_example }
```

`error.p` produces this error if you compile it without the –`config sparc` option:

```
hostname% pc -xl error.p
Wed May 31 17:10 1983 error.p

Line 12 :  %error 'Unknown architecture.'
E --------------------^---'Unknown architecture.'
Compilation failed
hostname%
```

## *The %exit Directive*

The `%exit` directive tells the compiler to stop processing the current Pascal source file.

### *Syntax*

%exit

### *Comments*

If the compiler encounters an `%exit` directive within an `include` file, it stops processing the `include` file, but continues processing the source file in which it is included. In effect, `%exit` is equivalent to an end-of-file marker.

When the compiler processes an `%exit` directive within an `%if` or `%ifdef` construct, it closes all `%if` or `%ifdef`s before it stops processing the current file.

### *Example*

The Pascal program, `exit_directive.p`:

```
program exit_directive(output);

begin
    writeln('Hello, world!')
end. { exit_directive }
%exit
Everything after the %exit is ignored.
So you can put anything here.
```

The commands to compile and execute the following: `exit_directive.p`

```
hostname% pc -xl exit_directive.p
hostname% a.out
Hello, world!
hostname%
```

## *The %if Directive*

The `%if` directive is a conditional branching directive.

### *Syntax*

```
%if expression %then
      .
      .
%end if
```

### *Comments*

When the compiler encounters an `%if` directive, it evaluates *expression*. If *expression* is true, the compiler executes the statements in the `%then` part. If *expression* is false, the compiler skips over the `%then` part and executes the `%else`, `%elseif`, or `%endif` directive or, if there is none, proceeds to the next statement.

The *expression* consists of a conditional variable and the optional boolean operators `and`, `or`, and `not`. You can set a conditional variable on the command line by using the `-config` option. See “Conditional Processing Variable: –config” on page 32 for information on this option.

Assuming `one` and `two` are conditional variables, *expression* may be any of the following:

one

two

one and two

one or two

not one

not two

### *Example*

See the example in the `%else` listing on page 200.

## *The %ifdef Directive*

The `%ifdef` directive determines whether or not you previously defined a conditional variable in a `%var` directive.Syntax

```
%ifdef expression %then
       .
       .
%elseifdef expression %then
       .
       .
%endif
```

### *Comments*

The *expression* consists of a conditional variable and the optional boolean operators `and`, `or`, and `not`. See the `%else` listing for examples of *expression*.

`%ifdef` is especially useful for determining whether or not a conditional variable has been declared in an `include` file.

### *Example*

See the example in the `%elseifdef` listing earlier in this appendix.

## *The %include Directive*

The `%include` directive inserts lines from the specified file in the input stream.

### *Syntax*

`%include '`*filename*`';`

### *Comments*

When `cppas` encounters the `%include` directive, it inserts the lines from the filename into the input stream.

### *Example*

The program unit, `include_prog.p`:

```
program include_prog;

%include 'extern.h';

begin
    global := 1;
    writeln('From MAIN, before PROC: ',global);
    proc;
    writeln('From MAIN,  after PROC: ',global);
end. { include_prog }
```

The module unit, `include_mod.p`:

```
module include_mod;

define
    global, proc;

%include 'extern.h';

procedure proc;

begin
    writeln('From PROC            : ',global);
    global := global + 1;
end; { proc }
```

The `include` file, `include.h`:

```
var
    global : integer;

procedure proc; extern;
```

The commands to compile and execute `ext_prog.p` and `ext_mod.p`:

```
hostname% pc -xl include_prog.p include_mod.p
include_prog.p:
include_mod.p:
Linking:
hostname% a.out
From MAIN, before PROC:1
From PROC       : 1
From MAIN,  after PROC:2
hostname%
```

## *The %list Directive*

The `%list` directive enables a listing of the program.

### *Syntax*

%list;

### *Comments*

The `%list` directive and the `-l` compiler option perform the same function.

### *Example*

The Pascal program, `list.p`:

```
program list_example(output);

{ This program demonstrates the use of the %list
  and %nolist directives. }

%list;
%include 'types.h';
%nolist;

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

The `include` file, `types.h`:

```
type
    lowints = 0..100;
    primary_colors = set of (red, yellow, blue);
    possibilities = set of boolean;
    capital_letters = set of 'A'..'Z';
    digits = set of lowints;

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

The listing includes the time each unit was compiled and the name of each unit compiled.

```
hostname% pc -xl list.p
Mon Jul 10 15:48 1989  list.p:
    6  %list;
Mon Jul 10 15:46 1989  ./types.h:
    1  type
    2     lowints = 0..100;
    3     primary_colors = set of (red, yellow, blue);
    4     possibilities = set of boolean;
    5     capital_letters = set of 'A'..'Z';
    6     digits = set of lowints;

    8  var
    9     pri: primary_colors;
    10    pos: possibilities;
    11    cap: capital_letters;
    12    dig: digits;
Mon July 10 15:48 1989  list.p:
    7  %include 'types.h';


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

## *The %nolist Directive*

The `%nolist` directive disables the program listing.

### *Syntax*

%nolist;

### *Comments*

`%nolist` is the default.

### *Example*

See the example under the `%list` directive earlier in this appendix.

### *The %slibrary Directive*

`cppas` treats `%slibrary` the same as the `%include` directive. See the `%include` listing earlier in this appendix.

### *The %var Directive*

The `%var` directive defines conditional variables for the preprocessor.

#### *Syntax*

`%var` *var1 ..., varN*

#### *Comments*

A conditional variable is defined when it appears in a `%var` directive; otherwise, it is undefined.

#### *Example*

See the example under the `%config` listing earlier in this appendix.

### *The %warning Directive*

The `%warning` directive tells the compiler to print a string on the standard output as a compiler warning.

#### *Syntax*

`%warning 'string'`

#### *Comments*

The compiler still produces an object file.

### *Example*

The Pascal program, warning.p:

```
program warning_example(output);

{ This program demonstrates the use of the
  %warning compiler directives. }

%var blue

begin
    %if blue %then
        writeln('The color is blue.');
    %else
        %warning 'Color not defined'
    %endif
end. { warning_example }
```

The output when you compile `warning.p` without the `-config` option:

```
hostname% pc -xl warning.p
Mon Dec 10 15:03 1993 warning.p

Line 12:%warning 'Color not defined'
w -------------------^----- 'Architecture not defined'
hostname%
```

# A

# *Compiler Error Messages* B

10010: Builtin <*function*> takes exactly <*number*> argumen*ts*

10020: Builtin <*function*> takes at least <*number*> arguments

10030: Builtin <*function*> takes at least <*number*> arguments and at most <*number*>.

10040: Built-in <*function*> cannot be passed as a parameter

10050: argv takes two arguments

10060: argv's first argument must be an integer, not <*type*>

10070: argv's second argument must be a string, not <*type*>

10080: Argument to card must be a set, not <*type*>

10090: flush takes at most one argument

10100: flush's argument must be a file, not <*type*>

10110: Not enough arguments to <*function*>

10120: Too many arguments to <*function*>

10130: Actual argument cannot be conformant array

10140: Actual argument is incompatible with formal var parameter <*identifier*> of <*function*>

10150: Actual argument is incompatible with formal <*paramtype*> parameter <*identifier*> of <*function*>

10160: Actual argument to NONPASCAL procedure cannot be conformant array

10170: Extra arguments to ADDR ignored

10180: Argument to <*function*> must be of type <*type*>, not <*type*>

10190: Second argument to <*function*> must be of type <*type*>, not <*type*>
10200: First argument to <*function*> must be of type <*type*>, not <*type*>
10210: Illegal argument to IN_RANGE
10220: Type clash in argument to IN_RANGE
10230: Illegal argument to ADDR
10240: Third argument to <*function*> must be of type <*type*>, not <*type*>
10250: Argument to ADDR is a dynamically allocated variable <*identifier*>
10260: Argument to ADDR is an internal variable <*identifier*>
10270: Argument to ADDR is a nested function <*function*>
10280: Argument to ADDR is an internal procedure <*function*>
10290: Fourth argument to <*function*> must be of type <*type*>, not <*type*>
10300: First argument to <*function*> cannot be a univ_ptr
10310: <*number*> argument to <*function*> must be of type <*type*>, not <*type*>
10320: <*number*> argument to <*function*> must be unpacked
10330: <*number*> argument to <*function*> must be packed
10340: <*function*> (line <*number*>) has <*number*> arguments
10350: Transfer functions take exactly one argument
10360: sizeof takes at least 1 argument
10370: Formal arguments should be given only in forward declaration
10380: Types must be specified for arguments
10390: Each procedure/function argument must be declared separately
10400: <*function*> takes no arguments
10410: <*function*> takes either zero or one argument
10420: <*function*> takes exactly one argument
10430: <*function*>'s argument must be integer or real, not <*type*>
10440: seed's argument must be an integer, not <*type*>
10450: <*function*>'s argument must be a real, not <*type*>
10460: <*function*>'s argument must be an integer or real, not <*type*>
10470: ord's argument must be of scalar type, not <*type*>
10480: <*function*>'s argument must be of scalar type, not <*type*>
10490: odd's argument must be an integer, not <*type*>
10500: chr's argument must be an integer, not <*type*>
10510: Argument to eoln must be a text file, not <*type*>
10520: Argument to eof must be file, not <*type*>

10530: Transfer functions take only one argument
10540: Arguments to <*function*> must be variables, not expressions
10550: Read requires an argument
10560: Write requires an argument
10570: Message requires an argument
10580: null takes no arguments
10590: <*function*> expects one argument
10600: Argument to <*function*> must be a file, not <*type*>
10610: <*function*> expects one or two arguments
10620: First argument to <*function*> must be a file, not <*type*>
10630: Second argument to <*function*> must be a string, not <*type*>
10640: <*function*> expects at least one argument
10650: (First) argument to <*function*> must be a pointer, not <*type*>
10660: Second and successive arguments to <*function*> must be constants
10670: Argument to <*function*> must be a alfa, not <*type*>
10680: halt takes no arguments
10690: stlimit requires one argument
10700: stlimit's argument must be an integer, not <*type*>
10710: remove expects one argument
10720: remove's argument must be a string, not <*type*>
10730: linelimit expects two arguments
10740: linelimit's second argument must be an integer, not <*type*>
10750: linelimit's first argument must be a text file, not <*type*>
10760: page expects one argument
10770: Argument to page must be a text file, not <*type*>
10780: Assert expects one or two arguments
10790: Second argument to assert must be a string, not <*type*>
10800: pack expects three arguments
10810: unpack expects three arguments
10820: Illegal transfer function argument
10830: constant argument expected
10840: Illegal argument with format radix specification; probably a comma missing
11010: have incompatible conformant array schemas

11020: sizeof(conformant array) is undefined
11030: Conformant arrays are not allowed at ANSI Level 0
11040: Subscripting allowed only on arrays, not on <*type*>s
11050: subrange value or array subscript (<*integer*>) is out of range
11060: compiler takes size of array
11070: Elements of a packed array cannot be passed by reference
11080: Subscripting allowed only on arrays and varying, not on <*type*>s
11090: Varying size must be a constant
11100: Size of <*identifier*> is zero
11110: Character array lower bound <> 1
11120: Character array upper bound of 1
11130: <*function*> requires a to be an unpacked array, not <*type*>
11140: <*function*> requires z to be a packed array, not <*type*>
11150: <*operation*> not allowed on arrays - only allow = and <>
11160: Packed multidimensional conformant arrays are not permitted
11170: For-statement variable <*identifier*> cannot be an element of a record
11180: . allowed only on records, not on <*type*>s
11190: <*identifier*> is not a field in this record
11200: Record required when specifying variant tags
11210: Missing case constants in variant record
11220: <*identifier*> is a duplicate field name in this record
11230: Duplicate variant case label in record
11240: <*operation*> not allowed on records - only allow = and <>
11250: Variable in with statement refers to <*identifier*>, not to a record
11260: Too many tag fields
11270: No variant case label value equals specified constant value
11280: Tag fields cannot be <*type*>s
11290: Bad variant constant
11300: variant required after case
12010: Assert expression must be Boolean, not <*type*>s
12020: illegal transfer function with <*identifier*>
12030: Illegal transfer function
12040: Transfer functions on bit fields not implemented

12050: Oct/hex allowed only on writeln/write calls

12060: Width expressions allowed only in writeln/write calls

12070: Cannot write <*type*>s with two write widths

12080: Can't write <*identifier*>s with oct/hex

12090: First write width must be integer, not <*type*>

12100: Second write width must be integer, not <*type*>

12110: Negative widths are not allowed

12120: Cannot write unpacked array to textfile

12130: Can't read <*type*>s from a text file

12140: Can't 'readln' a non text file

12150: Oct/hex allowed only on text files

12160: Write widths allowed only on text files

12170:  Cannot write unpacked array to textfile

12180: Can't write <*type*>s to a text file

12190: Can't 'writeln' a non text file

13010: constant identifier required

13020: Constant value cannot be evaluated at compile-time

13030: Constant too large for this implementation

13040: <*identifier*> is a constant and cannot be qualified

13050: Constant string too long

13060: Constant expression is required

13070: constant argument expected

13080: newline in string or char constant

13090: empty character constant

13100: too many characters in character constant

14010: Constant declarations should precede type, var and routine declarations

14020: Label declarations should precede const, type, var and routine declarations

14030: Type declarations should precede var and routine declarations

14040: All types should be declared in one type part

14050: All constants should be declared in one const part

14060: INTERNAL ignored, procedure was previously declared PUBLIC or as EXTERNAL.

14070: <*function*> has already been declared forward

14080: Unknown language <*identifier*> in EXTERN procedure declaration ignored

14090: Unresolved forward declaration of <*function*>

14100: <*identifier*> DEFINED, but not declared

14110: label <*identifier*> was declared but not defined

14120: PUBLIC procedures must be declared at outer block level

14130: PRIVATE ignored, procedure was declared DEFINED previously.

14140: PUBLIC ignored, procedure was declared INTERNAL or PRIVATE previously.

14150: PRIVATE ignored, procedure was declared PUBLIC or as EXTERNAL previously.

14160: For-statement variable <*identifier*> must be declared in the block in which it is used

14170: All labels should be declared in one label part

14180: Expected identifier VARYING in declaration.

14190: Variable declarations should precede routine declarations

14200: All variables should be declared in one var part

14210: public vars must be declared at outer block level

14220: Declarations may not be both STATIC and PUBLIC, STATIC is ignored

14230: Declarations may not be both DEFINE and PRIVATE, DEFINE is ignored

14240: Declarations may not be both EXTERN and PRIVATE, EXTERN is ignored

14250: <*identifier*> was declared in a DEFINE, cannot be STATIC

14260: <*identifier*> was declared in a DEFINE, cannot be PRIVATE

14270: <*identifier*> used as both a field and a type name in a record definition

14280: cannot DEFINE <*identifier*>, variable was not previously declared as EXTERN

14290: Declaration found when statement expected

14300: Expected keyword begin after declarations, before statements

14310: Improper initialization for variable <*identifier*>

14320: <*identifier*> is already defined globally.

14330: Definition of name <*identifier*> after applied use in <*identifier*>

14340: Definition of name <*identifier*> after applied use

14350: <*identifier*> is already defined in this block

14360: Range lower bound exceeds upper bound

14370: '*' subrange descriptor used in illegal context

14380: Cannot initialize dynamic local variables

15010: File <*identifier*> listed in program statement but not declared

15020: File <*identifier*> listed in program statement but declared as a <*identifier*>

15030: File <*identifier*> listed in program statement but defined as <*identifier*>

15040: Files cannot be passed by value

15050: Files cannot be a component of <*identifier*> passed by value

15060: Pre-defined files input and output redefined

15070: Files cannot be members of files

15080: Missing closing <*quote or bracket*> for include file name

15090: Include filename must end in .i or .h

15100: Cannot open #include file <*filename*>

15110: line <*number*> does not exist in file <*identifier*>

15120: End-of-file expected - QUIT

15130: Unexpected end-of-file - QUIT

16010: <*identifier*> is not a function

16020: Too many levels of function/procedure nesting

16030: No assignment to the function variable

16040: Functions should not return <*type*>s

16050: Procedure/function nesting too deep

16060: pcc_fvar(): no result for this function

16070: <*identifier*> is a <*class*>, not a function

16080: Illegal function qualificiation

16090: compiler takes size of function

16100: Can't call <*identifier*>, its  not a pointer to a procedure or function

16110: Can't qualify a function result value

16120: Cannot assign value to built in function

16130: INTERNAL option illegal for procedure pointer - option ignored.

16140: Unknown option for procedure pointer ignored : <*option*>

16150: Too many levels of function/procedure nesting

16160: Procedure/function nesting too deep

16170: Can't call <*identifier*>, its  not a pointer to a procedure or function

16180: Can't call <*identifier*>, it’s <*class*> not a procedure

16190: Procedure <*identifier*> found where expression required

16200: Illegal procedure call, expecting value

16210: Non-pascal routine <*identifier*> will fail if called indirectly from pascal

16220: Passing nested routine <*identifier*> to non-pascal routine <*identifier*>

16230: Can't call <*identifier*>, its not a pointer to a procedure or function

17010: Case label out of range

17020: Real constant out of range for this implementation

17030: Short real out of range for this implementation

17040: Short real <*number*> out of range for this implementation

17050: Implementation restriction: sets must be indexed by 16 bit quantities

17060: Subscript of <*identifier*> is out of range

17070: Subscript value of <*number*> is out of range

17080: subrange value or array subscript (<*integer*>) is out of range

17090: Successor of <*integer*> is out of range

17100: Predecessor of <*integer*> is out of range

17110: Value of <*integer*> is out of range

17120: Range upper bound of <*integer*> out of set bounds

17130: Range lower bound of <*integer*> out of set bounds

17140: Value of <*integer*> out of set bounds

17150: value of <*integer*> (initial assign. to for loop variable) is out of range

17160: Value of <*integer*> (terminal assign. to for loop variable) is out of range

17170: Tag type is out of range

17180: Base out of range (2..36)

18020: (* in a (* ... *) comment

18030: { in a { ... } comment

18040: Comment does not terminate - QUIT

18050: Illegal character, use the -xl option to process % compiler directives

18060: Illegal character

18070: unexpected EOF

18080: Point of error

18090: Parsing resumes

18100: Parse stack overflow

18110: Expression passed to UNIV formal <*identifier*> was converted to <*type*> of size <*number*> bytes

18120: 8 or 9 in octal number

18130: Number too large for this implementation

18140: <*operation*> is undefined for reals

18150: <*identifier*> cannot have more elements in z (<*integer*>) than in a (<*integer*>)

18160: <*identifier*> is an unimplemented extension

18180: Initializer must be a string

18190: Initialization string <*string*> is too long

18200: Expected 'options', not <*identifier*>.

18210: Unknown option ignored : <*identifier*>.

18220: Duplicate options specification : <*identifier*>.

18230: Unimplemented option ignored : <*identifier*>.

18240: <*identifier*> undefined on line<*number*>

18250: <*identifer*> improperly used on line<*number*>

18260: <*identifier*> is neither used nor set

18270: <*identifier*> is never used

18280: <*identifier*> is used but never set

18290: Length of external label for identifier <*identifier*> exceeds implementation limit

18300: Applied use of <*identifier*> before definition in this block

18320: <*operation*> is forbidden for reals

18420: Undefined <*identifier*>

18430: Undefined identifier

# B

18440: Improper <*identifier*> identifier
18450: Deleted <*token*>
18460: Replaced <*token*> with a <*token*>
18470: Replaced <*class*> id with a <*class*> id
18480: Inserted <*token*>
18490: Expected <*token*>
18500: Label <*identifier*> not defined in correct block
18510: Label <*identifier*> redefined
18520: <*identifier*> is undefined
18530: ^ allowed only on files and pointers, not on <*type*>s
18540: Pascal uses [] for subscripting, not ()
18550: Error occurred on qualification of <*identifier*>
18560: division by 0
18580: Variable required in this context
18590: Universal pointers may not be dereferenced
18600: Illegal format
18610: Unknown format radix <*identifier*> ignored, expecting HEX or OCT
18620: Expression required
18630: Unreachable statement
18640: Unrecoverable syntax error - QUIT
18650: Too many syntax errors - QUIT
18660: Input line too long - QUIT
18670: Unrecognizable # statement
18680: Include syntax error - expected ' or \" not found
18690: Absurdly deep include nesting - QUIT
18700: Too many levels of include nesting - QUIT
18710: Bad arbitrary base integer
18720: Bad base <*number*> number, <*number*>
18730: Digits required after decimal point
18740: Digits required before decimal point
18750: Digits required in exponent
18760: Space required between number and word-symbol
18770: Character/string delimiter is '
18780: Unmatched ’ for string

18790: Null string not allowed
19010: Module name expected after keyword MODULE
19020: Identifiers after module name ignored
19030: Missing program statement
19040: Input is used but not defined in the program statement
19050: Output is used but not defined in the program statement
19060: Program parameter <*identifier*> is repeated
20010: Left operand of <*operator*> must be integer, real or set, not <*type*>
20020: Right operand of <*operator*> must be integer, real or set, not <*type*>
20030: Cannot mix sets with integers and reals as operands of <*type*>
20040: Operands of <*operator*> must be Boolean or Integer
20050: Left operand of / must be integer or real, not <*type*>
20060: Right operand of / must be integer or real, not <*type*>
20070: Left operand of <*operator*> must be Boolean, not <*type*>
20080: Right operand of <*operator*> must be Boolean, not <*type*>
20090: Left operand of <*operator*> must be integer, not <*type*>
20100: Right operand of <*operator*> must be integer, not <*type*>
20110: Right operand of 'in' must be a set, not <*type*>
20120: Operands of + are of incompatible types
20130: Operand of <*operator*> must be integer or real, not <*type*>
20140: Operands of <*operator*> must both be sets
20150: Set types of operands of <*operator*> must be compatible
20160: Incorrect statement usage const. with operand
20170: ^ allowed only on files and pointers, not on <*type*>s
20180: <*operation*> is an illegal operation on strings.
20190: not must operate on a Boolean or Integer, not <*type*>
20200: not must operate on a Boolean, not <*type*>
20210: <*operation*> not allowed on pointers - only allow = and <>
20220: Strings not same length in <*operator*> comparison
20230: <*identifier*>s and <*identifier*>s cannot be compared - operator was <*operator*>
20240: <*identifier*>s may not participate in comparisons

21010: Attempt to pass IN parameter to formal reference parameter

21020: Expression type clashed with type of value parameter <*identifier*> of <*identifier*>

21030: Expression given (variable required) for var parameter <*identifier*> of <*identifier*>

21040: Parenthesis not allowed for var parameter <*identifier*> of <*identifier*>

21050: Parameter type not identical to type of var parameter <*identifier*> of <*identifier*>

21060: Packed/unpacked type conflict between formal and actual parameters

21070: Conformant array parameters in the same specification must be the same type.

21080: actual parameter is not an array

21090: array index type is incompatible with conformant array parameter <*identifier*>

21100: array index type is not within of index type of conformant array parameter <*identifier*>

21110: array index type is not within range[<*number*>..<*number*>] of index type of conformant array parameter <*identifier*>

21130: <*class*> <*identifier*> given for formal <*class*> parameter <*identifier*>

21140: does not match type of formal <*class*> parameter <*identifier*> (line <*number*>)

21150: <*class*> parameter <*identifier*> of <*identifier*> (line <*number*>)

21160: does not match <*class*> parameter <*identifier*> of <*identifier*> (line <*number*>)

21170: Type of <*class*> parameter <*identifier*> of <*identifier*> (line <*number*>)

21180:  does not match type of <*class*> parameter <*identifier*> of <*identifier*> (line <*number*>)

21190: Parameter congruity error:  incompatible groupings

21200: Packed/unpacked type conflict between parameters of <*identifier*> and <*identifier*>

21210: than formal <*class*> parameter <*identifier*> (line <*number*>)

21220: Conformant array parameter bound symbol table entry is NULL

21230: Program parameter <*identifier*> is repeated

21240: Previous declaration for formal parameter '<*identifier*>' has different type.

21250: Previous declaration for formal parameter '<*identifier*>' has different name '<*identifier*>'.

21260: Previous declaration for procedure '<*identifier*>' had different number of formal parameters.

21270: Formal <*class*> <*name*> cannot be qualified

21280: <*class*> <*name*> cannot be qualified

21290: Expression given, <*class*> required for <*class*> parameter <*identifier*>

21300: Variable given, <*class*> required for <*class*> parameter <*identifier*>

21310: Cannot take value of OUT parameter

21320: Invalid assignment to a parameter marked as IN

21330: Parameter congruity error: incompatible groupings

21340: UNIV parameter <*identifier*> should be passed as a var parameter

21350: Fields which select variant parts may not be passed by reference

21360: Cannot pass components of packed structures by reference

21370: Cannot pass components of packed structures by VAR, OUT, or IN/OUT

22010: Ran out of memory (gentypind)

22020: Ran out of memory (case)

22030: Ran out of memory (hash)

22040: Ran out of memory (TRIM)

22050: Ran out of memory (defnl)

22060: Ran out of memory (buffer_ir_pass)

22070: Ran out of memory (string)

22080: Ran out of memory (tralloc)

22090: out of memory (tstr)

22100: out of memory

22110: Ran out of hash tables

22120: Ran out of tree tables

22130: Out of space (put_on_idlist)

22140: out of tree space; try simplifying

22150: out of temporary string space

23010: <*identifier*> is a non-standard function

23020: <*identifier*> is a non-standard procedure

23030: Two argument forms of reset and rewrite are non-standard
23040: NIL values in const declaration are non-standard
23050: Set constants in const declaration are non-standard
23060: Expressions in const declaration are non-standard
23070: Routine Options are not standard
23080: External procedures and functions are not standard
23090: Separately compiled routine segments are not standard.
23100: UNIV parameters are non-standard
23110: IN parameters are non-standard
23120: OUT parameters are non-standard
23130: IN OUT parameters are non-standard
23140: <*identifier*> is a nonstandard function
23150: OTHERWISE clause in case statements is non-standard
23160: Ranges in case statements are non-standard
23170: Transfer functions are non-standard
23180: Reading scalars from text files is non-standard
23190: Writing <*type*>s with two write widths is non-standard
23200: Zero widths are non-standard
23210: Oct and hex are non-standard
23220: Writing <*type*>s to text files is non-standard
23230: <*identifier*> is a nonstandard procedure
23240: Short-circuit operators are non-standard
23250: <*operator*> comparison on sets is non-standard
23260: record comparison is non-standard
23270: Storage Class non-standard
23280: Initialization in declaration part is non-standard
23290: UNIV_PTR types are non-standard
23300: '_' in an identifier is nonstandard
23310: Octal constants are non-standard
23320: <*operator*> is non-standard
24010: Cannot exit -- not within a loop.
24020: For-statement variable <*identifier*> must be unqualified
24030: For-statement variable <*identifier*> cannot be an element of a record

24040: For-statement variable <*identifier*> may be illegally changed at line <*number*>

24050: For-statement variable <*identifier*> must be declared in the block in which it is used

24060: For-statement variable <*identifier*> cannot be <*type*>s

24070: Can't modify the for-statement variable <*identifier*> in the range of the loop

24080: Incorrect control variable

24090: Type of initial expression clashed with index type in 'for' statement

24100: Type of limit expression clashed with index type in 'for' statement

24110: Can't modify the for variable <*identifier*> in the range of the loop

24120: Case selectors cannot be <*type*>s

24130: Duplicate otherwise clause in case statement

24140: Case label type clashed with case selector expression type

24150: Multiply defined label in case, lines <*number*> and <*number*>

24160: No case-list-elements

24170: Bad case constant

24180: Range in case label must be in ascending order

24190: Maximum number of case selectors exceeded

24200: Cannot next -- not within a loop.

24210: Goto <*label*> is into a structured statement

24220: Goto <*label*> from line <*number*> is into a structured statement

24230: Variable in with statement refers to <*type*>, not to a record

24240: Maximum WITH nesting depth exceeded, ignoring WITH variables

24250: Variable in with statement not correct

24260: cannot assign to lhs

24270: Variable required

24280: <*class*> <*identifier*> found where variable required

24290: univ_ptr variable cannot be dereferenced in assignment

24300: Improper use of the DEFINE statement.

24310: End matched <*keyword*> on line <*number*>

24320: Inserted keyword end matching {begin|record|class} on line <*number*>

25010: {pack|unpack}: elements of a and z must have the same type

25020: component types of formal and actual arrays are not conformable

25030: Type of function <*name*> (line <*number*>)

25040: Case label type clashed with case selector expression type

25050: Type clash: initializer must be a pointer

25060: Type clash: type of initializer does not match type of array

25070: Type clash: Integer is incompatible with real

25080: Type clash:  Initializer out of range

25090: Type clash: real is incompatible with integer

25100: This resulted because you used '/' which always returns real rather

25110: than 'div' which divides integers and returns integers

25120: Type clash: non-identical scalar types

25130: Type clash: unequal length strings

25140: Type clash: varying char and unknown

25150: Type clash: packed and unpacked set

25160: Type clash: files not allowed in this context

25170: Type clash: non-identical <*class*> types

25180: Type clash: <*type*>s with file components not allowed in this context

25190: Type clash: <*type*> is incompatible with <*type*>

25200: Type clash: string and unpacked array

25210: Type clash: string and packed array with lower bound <> 1

25220: Type clash: Non-identical string types must be packed

25230: Type clash: packed array with lower bound <> 1

25240: Set default type 'intset' is not a set

25250: Upper bound of element type clashed with set type in constant set

25260: Lower bound of element type clashed with set type in constant set

25270: Element type clashed with set type in constant set

25280: Set elements must be scalars, not <*type*>s

25290: Set of real is not allowed

25300: Procedures cannot have types

25310: Function type can be specified only by using a type identifier

25320: Function type should be given only in forward declaration

25330: Different type declared previously for function return <*name*>.

25340: Function type must be specified

25350: Type of expression in while statement must be Boolean, not <*type*>

25360: Until expression type must be Boolean, not <*type*>, in repeat statement

25370: Array index type incompatible with declared index type

25380: Too many subscripts

25390: Bad type of variable used for call

25400:  Transfer functions only work on types of the same size

25410: Only type char allowed in varying array

25420: Type mismatch in read from non-text file

25430: Type mismatch in write to non-text file

25440: Specified tag constant type clashed with variant case selector type

25460: Tag type is out of range

25470: Variant label type incompatible with selector type

25480: set types must be compatible in comparisons - operator was <*operator*>

25490: <*class*> types must be identical in comparisons - operator was <*operator*>

25500: Index type clashed with set component type for 'in'

25510: Operands of + are of incompatible types

25520: Type names (e.g. <*type*>) allowed only in declarations

25530: Type name (e.g. <*type*>) found in illegal context

25540: Set types of operands of <*operator*> must be compatible

25550: expression has invalid type

25560: Type of expression clashed with type of variable in assignment

25570: Type of expression in if statement must be Boolean, not <*type*>

25580: Type of transfer function does not match type of right side

25590: <*identifier*> is a <*class*>, not a type as required

25600: Set type must be range or scalar, not <*type*>

25610: Sets must have basetype with range <*number*>..<*number*>

25620: Subrange type cannot contain a expression

25630: Scalar types must be identical in subranges

25640: Can't mix <*identifier*>s and <*identifier*>s in subranges

25650: Subrange of real is not allowed

25660: Subrange bounds must be Boolean, character, integer or scalar, not <*type*>

25670: Index type for arrays cannot be real

25680: Array index type is a <*type*>, not a range or scalar as required

25690: Packed conformant array schema requires type identifier as element type

25700: Illegal type in VARYING array, only type CHAR allowed

25710: Size of <*type*> type exceeds implementation limits

25720: Initializer must be a constant value of same type

26010: Malformed program statement

26020: Malformed declaration

26030: Malformed const declaration

26040: Malformed type declaration

26050: Malformed var declaration

26060: Malformed record declaration

26070: Malformed statement in case

26080: Malformed statement

26090: Missing/malformed expression

26100: Label restricted to 4 digits

26110: Label declared non-integer

26120: Replaced unknown parameter specifier <*identifier*> with OUT

26130: Deleted ';' before keyword else

27010: Integer overflow in constant expression

Extension to WITH statement not allowed

## *Internal Error Messages*

The following are internal error messages and should not occur:

18330: NAME with no symbol table entry

18340: ir_symbol: unknown class

18350: ir_leaf: cannot take address of op *<identifier>*

18360: ir_leaf: illegal leaf node, op = *<identifier>*

18370: ir_starg: illegal STARG

18380: ir_exit: no return label

18390: cannot take address of op <identifier>

18400: op '<identifier>' not implemented yet

18410: goal <integer> not defined for op <operation>

18570: wasted space: <number>

# B

# *Index*

## D



## E

## F



## G

## H



## I



## K



## L

## T



## U



## V

## W



## X



## Z