# C

## A Programming Language For The Fortune 32:16

FORTUNE SYSTEMS
300 Harbor Boulevard
Belmont, CA 94002

# LANGUAGE DEVELOPMENT TOOLS GUIDE
# RELEASE 1.8
# ADDITIONS AND CHANGES

This document contains important information for the Language Development Tools Guide, and pertains to Release 1.8 of the Language Development Tools set and the language compilers C, Pascal, and FORTRAN.

## SOFTWARE EXCEPTIONS

When running a program compiled with the profiler option (-p), do not rely on the exit status returned by that option. Exit status of the profiled program may be checked at runtime.

FORTUNE SYSTEMS
101 Twin Dolphin Drive
Redwood City, CA 94065

R E L E A S E

# LETTER

Dear Fortune Systems Customer:

This package contains the latest release of the Fortune C Language. You should find the following items included in this package:

- Fortune C Language Master Diskette (Release 1.7)
- *Fortune C Language Guide* and index tab
  (The guide and tab can be inserted into the Language Development Tools binder.)
- *The C Programming Language*, by B.W. Kernighan and D.M. Ritchie
- User Response Card
- Software Registration Card

If any item is missing, contact a Fortune Representative. Please be sure to fill out and return the software registration card.

C is a general purpose programming language that is often described as a "powerful assembly language." It offers the programmer the advantages of coding brevity, variety of data structures, modern flow-control constructs, fast floating-point, single and double processor, and operators. C is well suited to system software development (most UNIX™ operating systems and Fortune's FOR:PRO are written in C) and has been used successfully in a wide range of commercial, scientific, and data base applications.

A copy of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, is also included in this package to aid further understanding of C programming.

In addition to C, Fortune Systems Corporation offers two other systems languages for development purposes: Pascal and FORTRAN. These packages identify Fortune extensions to the language and provide numerous programming examples.

If you do not have Fortune Language Development Tools (part number 1002145-02) installed on your system, you will need to order this package separately before installing the Fortune C Language. The Language Development Tools package is a language-independent companion document to the guide that accompanies the compiler disks. It is designed to meet the needs of experienced programmers for programming in the FOR:PRO environment.

We at Fortune Systems Corporation hope you find the Fortune C Language package to be as useful and effective as we do. We welcome your comments and suggestions concerning this and other Fortune products.

# *Fortune C Language Guide*

FORTUNE SYSTEMS
300 Harbor Boulevard
Belmont, CA 94002

Printed in the U.S.A.
1 2 3 4 5 6 7 8 9 0

**Ordering the Fortune C Language Guide**

Order No.: 1002175-01 for the complete guide with disk
1002147-01 for the complete guide without disk

Consult an authorized Fortune Systems Representative for copies of manuals and technical information.

**Disclaimer of Warranty and Liability**

No representations or warranties, expressed or implied, of any kind are made by or with respect to anything in this manual. By way of example, but not limitation, no representations or warranties of merchantability or fitness for any particular purpose are made by or with respect to anything in this manual.

In no event shall Fortune Systems Corporation be liable for any incidental, indirect, special or consequential damages whatsoever (including but not limited to lost profits) arising out of or related to this manual or any use thereof even if Fortune Systems Corporation has been advised, knew or should have known of the possibility of such damages. Fortune Systems Corporation shall not be held to any liability with respect to any claim on account of, or arising from, the manual or any use thereof.

For full details of the terms and conditions for using Fortune software, please refer to the Fortune Systems Corporation Customer Software License Agreement.

# *Contents*

# *Introduction*
# *An Overview of C*

The Fortune Systems' version of C is a general purpose programming language based on the C language developed by Bell Laboratories. C is fast and versatile. It is suitable for applications programming as well as for developing operating systems and compilers. The FOR:PRO operating system, the UNIX operating system on which C is based, and the C compiler were all written in C. The C language is also appropriate for numerical, text handling, and data base applications.

## THE C VOCABULARY

The major elements of the C vocabulary are:

| | |
|---|---|
| **Functions** | Logically grouped events. |
| **Declarations** | Areas of the program that define a section of code to be activated by function calls from other parts of the program. |
| **Arguments** | Methods of passing data to functions. |
| **Statements** | Parts of the program that determine action. |
| **Return Values** | Values returned by each function. |
| **Types** | Each declared variable is of a particular type. |
| **Structure** | Groups of logically associated data. |
| **I/O Functions** | Functions available through a library file that is linked automatically. |
| **Memory Management Functions** | Library routines that allocate and free space in the main memory. |
| **Math Functions** | A library of math functions. |
| **String Functions** | A library of string functions. |

## THE C PROGRAM

The basic structure of a C program is shown in Figure i-1.

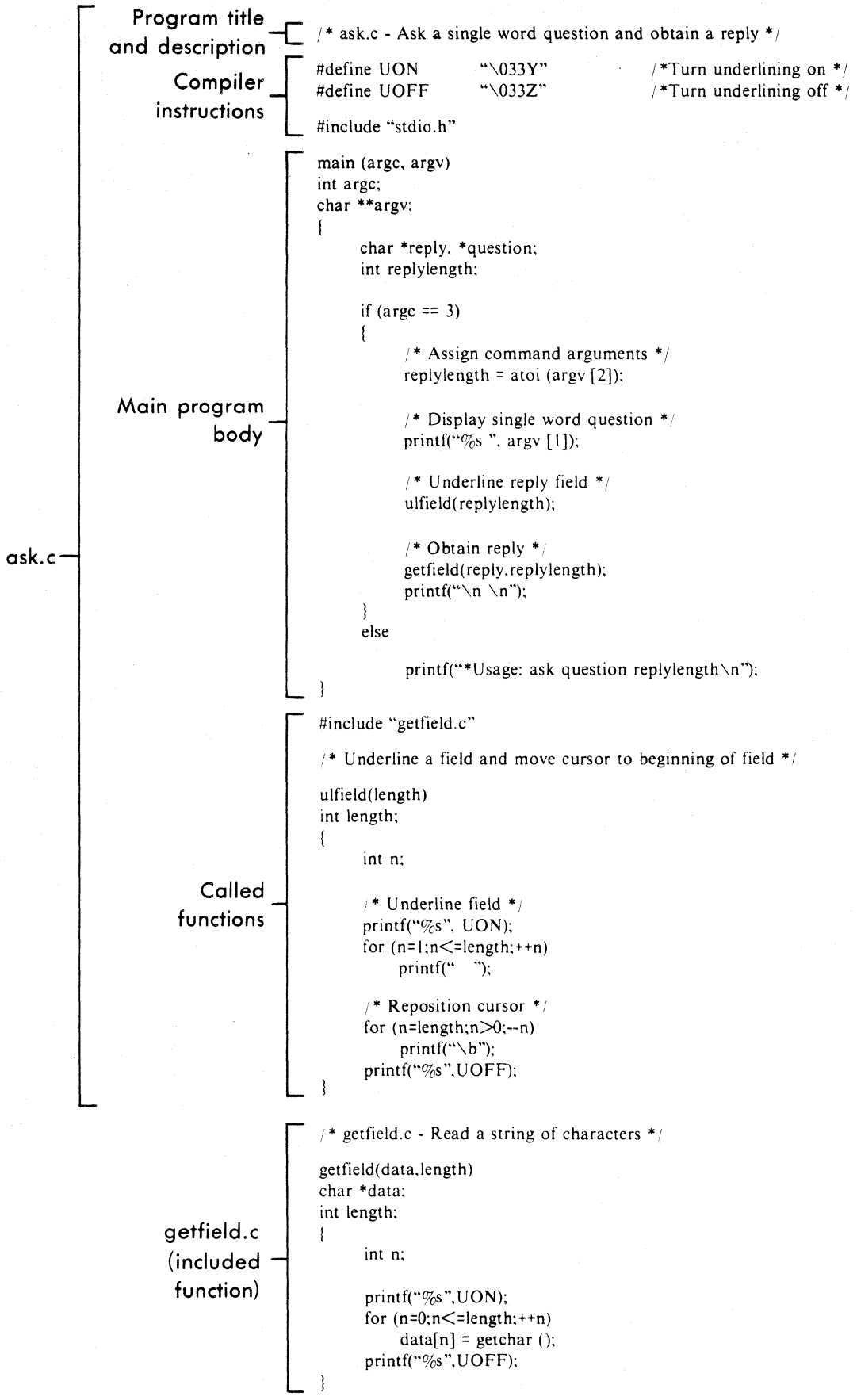


Figure i-1. Major Sections in a Typical C Program

## FORTUNE EXTENSIONS TO THE C LANGUAGE

Fortune Systems Corporation has included several extensions to the C language. These are:

- **Non-unique structure elements**

  Previously, all declarations at the same context level had to be unique. This extension permits non-unique structure element declarations that do not share the same offset value.

- **Long variable names**

  The Fortune C language permits variables to be of any length.

- **The void type**

  This type may be used to prevent "incompatible type" warning messages.

- **Passing of structure arguments by value**

  This feature of the Fortune version of C allows structures to be passed to functions by "value" as well as by the standard "reference."

- **Structures as function return values**

The Fortune C language permits structures to be returned as the value of a function.

These extensions are described in detail in Part 2, "The C Language."

## CONVENTIONS USED IN THIS GUIDE

When creating and executing C programs, you'll be communicating with the FOR:PRO operating system. FOR:PRO requires that commands be typed using proper command format, or syntax. This document uses certain conventions to illustrate how FOR:PRO commands are to be typed. These are outlined below.

- **Commands** to FOR:PRO are almost always lowercase letters, as is with most UNIX commands. Occasionally, some command options may be uppercase. Be careful to type commands using the case shown in the syntax descriptions.

- In syntax statements, any **input** that you must type is shown in **boldface.** Examples and filenames are always in boldface.

- Words that you must replace with your own text are underlined. Such items are also referred to as command-line parameters.

- **Brackets** [ ] indicate one or more options you may select from.

- **Hyphens** (-) usually signal an option on a command line, as in the command

  ```
  ls -l
  ```

  Always remember to type the hyphen.

- **Ellipses** (...) mean that the preceding option may be repeated.

- Commands are sent to FOR:PRO by pressing the RETURN key.

- This guide assumes you are using the standard Bourne shell, hence the $ prompt indicating the start of a command line. However, this is merely a convention. The FOR:PRO command syntax used throughout this guide applies to the C shell as well.

BIBLIOGRAPHY

Hancock, Les and Morris Krieger, The C Primer. New York: McGraw-Hill Book Company, 1982.

Kernigan, Brian W. and Dennis M. Ritchie, The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

Kochan, Stephen G., Programming in C. Hasbrouck Heights, New Jersey: Hayden Book Company, Inc.

# PART ONE
# USING C ON THE FOR:PRO SYSTEM

Part 1 of the Fortune C Language Guide introduces five fundamental steps in developing and using a C program on the FRO:PRO system. These are:

- Compiling a C program
- Linking a C program
- Executing a C program
- Debugging a C program
- Profiling a C program

# *Chapter 1*
# *Compiling a C Program*

The C compiler translates C source code into relocatable object code. It translates each file of code individually and then calls the linking loader to create a single file of object code.

## ENTERING SOURCE CODE

Generally, C source code is entered as a source file using a text editor. You can use the FOR:PRO line editor **ed** or any other editor you have on your system. For instructions on the use of the **ed** editor, see Chapter 1 of the Language Development Tools.

## INVOKING THE COMPILER

For input, the compiler expects one or more file names, each with the suffix **.c**. The compiler will not accept files without this suffix.

To invoke the C compiler, use the **cc** command with the following syntax:

```
$ cc [-options] file.c  file2.c ... filen.c
```

The **cc** command calls up to six different programs to translate a C source file into object code. They include, the following:

- Preprocessor
- C compiler
- Optimizer
- Fast floating-point processor
- Assembler
- Linking loader

These programs are called in successive passes. The flow chart in Figure 1-1 shows the progress of a C file through the compiler passes. Appendix A describes the passes in detail.

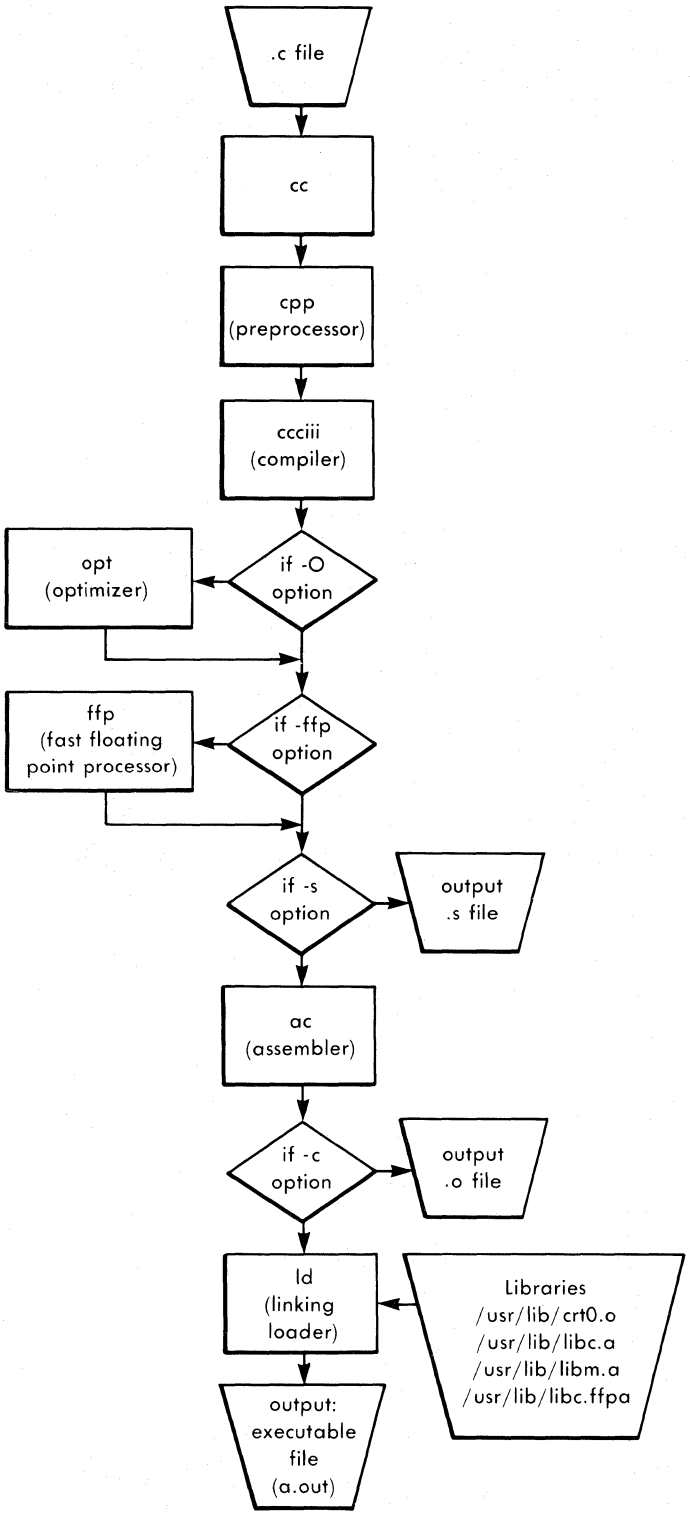


Figure 1-1. Path of a C Source File Through the Compiler

## SELECTING COMPILER OPTIONS

The output of the **cc** command is by default an executable object file called **a.out.** It is created in the current working directory. However, you may select a number of options that change the output of the compiler or alter the way it runs. Use the following syntax to select a compiler option:

```
$ cc -O file1.c
```

where file1.c is the C source file, and O is the option. Note that a hyphen - must precede all options.

Table 1-1 briefly describes the options of **cc** in alphabetical order for convenient reference. More detailed descriptions of the options appear in Appendix A.

## OBTAINING SOURCE LISTINGS

The C compiler does not generate source or cross reference listings. To obtain these, use the separate FOR:PRO programs **cat, more,** and **lpr.**

### cat: Listing Source Files

The **cat** program concatenates and displays files. You may use it to list a C source file on the terminal screen as follows:

```
$ cat file1.c
```

You can generate line numbers in the listing by inserting the **-n** option between the **cat** command and the filename:

```
$ cat -n file.c
```

### more: Viewing Long Program Listings

If a program is more than one screenful in length, you may wish to use the **more** command, since the text will scroll continuously if you use the **cat** command. The syntax for **more** is

```
$ more file.c
```

### lpr: Listing Source Files on a Printer

Finally, to send source files to a line printer, use the **lpr** command:

```
$ lpr file.c
```

Table 1-1. C Compiler Options

| Option | Purpose |
|---|---|
| **-c** | Creates a relocatable object file (.o suffix). |
| **-g** | Produces additional symbol table information for the Fortune symbolic debugger. |
| **-o** filename | Names the output executable file. |
| **-p** | Prepares the executable program for use with the profiler (prof). |
| **-v** | Lists the compiler passes as they occur. |
| **-w** | Suppresses warning messages. |
| **-C** | Prevents the preprocessor from removing comments. |
| **-D** name=def | Defines an identifier name to the preprocessor, in the same way as a **#define** macro statement. If no definition (= def) is given, name is defined as **1**. |
| **-E** | Runs only the preprocessor; output goes to standard output. |
| **-G** | Prevents the compiler from generating calls to stack checking routines at the beginning of each procedure. |
| **-I dir** | Searches for **#include** files in the **dir** directory. |
| **-O** | Calls the peephole optimizer. |
| **-S** | Generates assembly language output; output is left in **.s** file. |
| **-U** name | Eliminates the definition of the identifier name to the pre-processor. |

# *Chapter 2*
# *Linking a C Program*

The last pass of the C compiler is the linking loader. The linking loader combines files of object code, resolves external references, and searches libraries as necessary. It is invoked automatically by the **cc** command unless you use the **-c** compiler option.

If you are developing a large program, it is helpful to divide it into smaller programs and compile them separately using the **-c** option. When they have all been compiled successfully into object code, they can be linked together using the linking loader as follows.

## INVOKING THE LINKING LOADER

There are two ways to invoke the linking loader. The simplest method is to use the **cc** command:

```
$ cc [options] file.o file2.o file3.o
```

With this command, the three object code files--file.o, file2.o, and file3.o--are linked into one executable file, **a.out**.

### Sample Program for Use with the cc Command

> **NOTE:** Although this is a short example, separate compilation is most useful with long programs.

You have two files, **f1.c** and **f2.c**. The contents of **f1.c** are

```
main()
{
int x,y,z;
x=10;
y=20;
z=57;
bingo(x,y,z);
}
```

The contents of f2.c are

```
bingo(a,b,c)
int a,b,c;
{
printf("%d %d %d", a,b,c);
}
```

The **cc** command is used with the **-c** option to produce two files of object code:

```
$ cc -c f1.c
$ cc -c f2.c
```

or simply

```
$ cc -c f1.c f2.c
```

When f1.c and f2.c are successfully compiled, you will have two object code files, f1.o and f2.o. You link the files with the command

```
$ cc  f1.o f2.o
```

and one executable file **a.out** is produced.

The second way to invoke the linking loader is with the **ld** command. This command is not as simple to use as **cc**. With the **ld** command, you must call the individual libraries; the **cc** command calls them for you automatically. However, **ld** does make available several options that cannot be used with **cc**. These are shown in Table 2-1.

The syntax of the **ld** command is

```
$ ld [options] [/usr/lib/crt0.o]  file.o file2.o -lx -lx2
```

The snytax of the ld is

```
$ ld [options] [/usr/lib/crt0.o] file.o file2.o [-lx][-lx2]
```

In this command, **-lx** is an abbreviation for a library named **libx.a** in a given list of directories. The loader searches for matching libraries in the directory **/lib**, **/urs/lib**, and **/usr/local/lib**, in that order. To load the standard C library (**/usr/lib/libc.a**), include the flag **-lc** at the end of the loader command line.

The module **/usr/lib/crt0.o** is the standard runtime stuntoff module. It must be the first module in the command line if you want the loader output to be an executable command. (The **cc** command does this automatically.)

Table 2-1. The **ld** Options

| Element | Purpose |
|---|---|
| **-d** | Forces a definition of common storage. |
| **-e** | Specifies an alternate entry point. |
| **-n** | Indicates read-only outfile, so that text portion is shareable among users. |
| **-o filename** | Causes executable file produced by loader to be placed in **filename.**. (Default name: **a.out.**) |
| **-u symbol** | Takes the symbol and enters it undefined in a symbol table. |
| **-r** | Generates relocatable object code file. |
| **-s** | Removes symbol table and relocation bits from output. |
| **-D arg** | Takes argument as hexadecimal; pads data segment with 0 to indicate length. |
| **-M** | Produces primitive load map, listing files to be loaded. |
| **-N** | Makes text portion unsharable. |

The libaries contain object code that will resolve calls to the standard C routines (such as printf), calls to math functions, or other special subroutine calls. It is important that the standard C library **libc.a** be the last argument on the **ld** command line, since all other modules (and other libaries) reference subroutines defined in **libc.a**.

Some of the libraries you may need with the C programs and **ld** command are:

- /crt0.o Runtime start-off library
- /libc.a C library
- /libm.a Math library
- /libtermcap.a Library of terminal independent cursor movements

- /libcurses.a Library of screen functions (see Curses(3)).

If you use the **-ffp** option, you will not need the math library but you will need:

- libffp.a the floating-point library
- libc_ffp.a the C floating-point library

Below is an example of a local line for object modules compiled using the **-ffp** option:

```
$ ld -u_cforce /usr/lib/crt0.o mod1.o mod2.o -lc_ffp -lffp -lc
```

The **-u_cforce** is required to force loading of the **-ffp** modules in the C fast floating-point libraries rather than the standard library routines, which may not be compatible with **-ffp.** The **-ffp** option causes the -u option and arguments to be passed to the loader automatically.

## LINKING FILES IN DIFFERENT STAGES OF COMPILATION

You can combine files that are in different stages of compilation with the **cc** command. The 68000 assembly language files (suffix .s), object code files (suffix .o), and C source code files (suffix .c) can be compiled into one **a.out** file as follows:

```
$ cc file.s file2.o file3.c
```

This command converts the two files that are not already in object code (file.s and file3.c) into object code, resulting in file.o, file2.o, and file3.o. Because the linking loader was not suppressed, (that is, the **-c** option was not used) the three object code files are then linked, producing one executable file **a.out.**

## LINKING MODULES WRITTEN IN OTHER LANGUAGES

Source files written in languages other than C must be compiled using the **-c** option before they can be linked.

For example, to compile a C program with a program written in Pascal, use the **pc** command with the **-c** option. The **pc** command invokes the Pascal compiler, and **-c** suppresses the linking loader, resulting in an object code file of the Pascal program. The command is

```
$ pc -c pascfile.p
```

It produces file pascfile.o.

To link the object code file produced by the above command with a C file, you use:

```
$ pc -c pascfile.p
```

It produces file pascfile.o.

To link the object code file produced by the above command with a C file, you use:

```
$ cc pascfile.o Cfile.o
```

Remember that the C file and all other files in the **ld** command line must be in object code.

## HEADER FILES AND GLOBAL VARIABLES

Separate compilation of programs can save time when several program modules constitute one large program. Each module can be compiled separately and then they can be linked into one program with the linking loader.

Two techniques should be considered when performing separate compilations. First, a common set of definitions or declarations can be shared among the modules of a large C program. Rather than placing a set of these definitions in each module, you should declare them in a common header file. By convention, the suffix **.h** is appended to this header file name. Use the **#include** statement with the name of the header file as its argument in each file that requires the definitions.

The following example shows how to prepare files for separate compilation without duplicating the definitions:

```
file1.c:          file2.c:

 int i[100];       int i[100];
 int j[200];       int j[200];
 int k[100];       int k[100];

 func1()           func2()
 {                 {
 ...               ...
 ...               ...
 }                 }
```

Use **#include** statements with a header file:

```
defs.h:

int i[100];
int j[200];
int k[100];

file1.c:                file2.c:

#include "defs.h"       #include "defs.h"

func1()                 func2()
{                       {
...                     ...
...                     ...
}                       }
```

The second technique concerns global variables that are used across several C source files but are not defined in a common header file. In all files other than the one in which an external variable is defined, there must exist an external declaration. In the following example, this declaration permits the compiler to determine the type of a variable that is not defined in the module.

```
file1.c:            file2.c:

long val;           extern long long val;

main()              func()
{                   {
...                 ...
...                 ...
}                   }
```

# *Chapter 3*
# *Executing a C Program*

A file that has been compiled and linked can be executed. To execute a file, type its name and press RETURN. If you used the **-o** option in the **cc** command, the executable file's name was indicated with the option. If the **-o** option was not used, the executable file is named **a.out**. For example:

| | |
|---|---|
| $ **filename** | If the **-o** option was used during compilation |
| $ **a.out** | If the **-o** option wasn't used |

Errors that were not detected during compilation sometimes surface during program execution. These are called runtime errors. Sometimes runtime errors cause execution to stop with an error message. Often, runtime errors can only be found by examining the output or "side effects" of execution. Chapter 4 contains more information about debugging your programs.

## REDIRECTING INPUT

The standard input device is the keyboard. The C routines that access files for standard input are discussed in Chapter 11, "Input and Output."

In addition, it is possible to redirect input so it comes from a file instead of the keyboard. Use the redirection input symbol < to do this. The redirection symbols are shell commands, and are used on the shell command line. Here is the syntax used to execute **a.out** with redirected input.

```
$ a.out < datafile
```

The object file a.out will be executed and the contents of datafile, a file you have created in your directory, will be read into **a.out** when the appropriate C system call is reached.

## REDIRECTING OUTPUT

The standard output is the terminal screen. Chapter 11, "Input and Output," discusses the C system calls for generating output on the screen.

In addition, output may be redirected to a file or to the line printer.

## Redirecting Output to a File

Use the FOR:PRO redirection symbol > to redirect output to a file. Both the output of a successfully executed program and the error messages can be redirected to a file in your directory. For both, use >. To use > with **a.out**, type

```
$ a.out > store
```

This command will place the output of **a.out** in a file called **store**. If **store** contains any contents already, the old contents will be replaced by the new. To have the new data appended instead, use the double arrow >> as follows:

```
$ a.out >> store
```

The output of **a.out** will be added to the end of the file store.

The input and output redirection symbols may be used together in one command line to redirect both input and output:

```
$ a.out < infile > outfile
```

This command line inputs the contents of infile to a.out and places the output of **a.out** in outfile.

## Redirecting Output to the Printer

Two ways are available to send output to the printer. If the output of an executed program is stored in a file, the output file may be sent to the line printer spooler with the **lpr** command, which has the following syntax:

```
$ lpr [-options] outfile
```

Note that **lpr** is the FOR:PRO command that invokes the line printer spooler. The spooler ensures that one job is completely printed before another is started.

If output is normally sent to the standard output, it can be redirected with the pipe symbol | as in:

```
$ a.out | lpr
```

## HANDLING ERRORS

The C compiler error messages are designed to be self-explanatory and to help the user locate the line number in the source file where the error occurs. For example, if the compiler discovers the following **for** statement on line 5 of this sample C program:

```
main () { int jack;
jack = 0;
     for(i=0; i <=100; i++)
          jack = jack + i;
}
```

the error message is

```
"programname.c", line5: i undefined
```

Normally the display screen receives error messages even when standard output has been redirected to some other output device. This convention has been provided so that error messages will not be intermingled with other output.

Error messages are normally sent to device number 2, standard error. To redirect your error messages to a file, use the following command from the Bourne shell:

```
$ a.out > erfile 2 > &1
```

In the C shell use

```
$ a.out > &erfile
```

If you do not know which shell you are in, use the **set** command. If the variable **argv** is listed, you are in the C shell. Otherwise, you are in the Bourne shell.

The **fprintf** statement can also redirect error messages, using the predefined stream pointer **stderr** just as standard output uses **stdout.**

Error messages can be redirected from within a C program in this way:

```
$fprintf(stderr, "Error number %d\n", ernumber);
```

# *Chapter 4*
# *Debugging a C Program*

The Fortune sybmolic debugger **(fdb)** is one of the tools included with the Language Development Tools. The debugger is used when a program compiles successfully but does not run as expected. From within the debugger, you can trace variables and their values throughout a program to find the exact location of a bug.

This chapter describes the use of the Fortune symbolic debugger.

## COMPILING A PROGRAM FOR DEBUGGING

Before a file can be input to the debugger, the file must be compiled with the **-g** compiler option. This is done with the command:

```
$ cc -g file.c
```

## INVOKING THE DEBUGGER

After compiling the program with the **-g** option, you will have an executable file. As always, the file is called **a.out** unless you used the **-o** option. The next step is to run the **fdb** program with your executable file as follows:

```
$ fdb [filename] [directory]
```

where filename is the name of your executable file. If no filename is given, the file **a.out** is used by default. The directory, if included, refers to the directory where **filename** can be found. If no directory is named, the current working directory is assumed.

## USING THE **fdb** Commands

Table 4-1 defines the most important **fdb** commands. Additional commands as well as the general rules for using the debugger are described in the Language Development Tools Guide. These commands may be abbreviated.

Table 4-1. The fdb Commands

| Command | Function |
|---|---|
| **alias** | Defines or cancels an alias. |
| **break** | Sets a breakpoint. |
| **delete** | Removes a breakpoint. |
| **dump** | Dumps memory contents. |
| **file** | Changes input, output, and source files. |
| **find** | Searches source file for a string. |
| **fun** | Defines a function key. |
| **go** | Starts or resumes program execution. |
| **help** | Displays a summary of **fdb** commands. |
| **let** | Modifies a data item's value in memory. |
| **print** | Prints specified lines of the source code. |
| **quit** | Exits **fdb**. |
| **restart** | Restarts program with optional parameters. |
| **set** | Sets special debugger options. |
| **show** | Displays debugger status information or lines of source code. |
| **trace** | Traces program execution. |
| **walk** | Executes program line by line, stopping after each command. |
| **!** | Shell escape. |
| ,(comma) | Displays contents and addresses of a variable. |
| **&** | Displays address of a variable. |
| **return key** | Repeats previous command. |

## A SAMPLE C PROGRAM FOR DEBUGGING

The following C language program converts integers to ASCII characters. It can be used as an exercise in debugging.

To begin your debugging session, first create a file of the sample C program. Then, after compiling the file with the -g option, give the **fdb** command, and type the different debugger commands to see how the debugger works. Name the program "sample.c" and type it as file in your current working directory as follows:

```
#define  MAXCHAR   10
char s[MAXCHAR]
main()          /* sample.c */
{
     int     i;

     i= 12345;
     printd(i);
     i= -987654;

     printd(i);
}

printd(n)
int n;
{
     register    i;

     if (n < 0)  {
          putchar('-');
          --n;
     }
     i = 0;
     do  {
          s[i++] = n % 10 + '0';
     }  while ( (n /= 10) > 0 );
     printit(i);
}

printit(k)
int     k;
{
     while(--k >= 0)
          putchar(s[k]);
     putchar('\n');
}
```

Next, compile the C program with the **-g** option:

```
$cc -g sample.c
```

Then invoke the Fortune symbolic debugger with the command:

```
$fdb a.out
```

Your screen will show the message:

```
Fortune Symbolic Debugger
```

An asterisk (*) prompt will appear on your screen. Now follow the guide below. The information that you type is in boldface type. The purpose of the command is in parentheses.

```
* break 11
```

(Sets breakpoint at line 11.)

```
BREAKPOINT SET AT: PROCEDURE  main
LINE   11:         printd(i);
```

```
* break printd: 1 do ,n
```

(Sets breakpoint at the first line of procedure printd and display the value of n whenever the program stops.)

```
NEW BREAKPOINT SET AT: PROCEDURE  printd
LINE   19:             if (n < 0)  {
```

```
* break printit: 34
```

(Sets breakpoint at printit, line 34.)

```
NEW BREAKPOINT SET AT: PROCEDURE  printit
LINE   34:             putchar(s[k]);
```

```
* show break
```

(Displays the 3 breakpoints defined so far.)

```
PROCEDURE  printit
LINE   34:             putchar(s[k]);
PROCEDURE  printd
LINE   19:     if (n < 0)  {
BKPT COMMAND: ,n
PROCEDURE main
LINE   11:     printd(i);
```

```
* go
```

(Starts program execution and stops at the first breakpoint.)

```
INFORM(36): EXECUTION SUSPENDED TO
PERFORM BREAKPOINT COMMAND ,n AT
printd:LINE   19:      if (n < 0)  {
12345

* show window 7
```

(Displays 7 lines before and 7 lines after the current line.)

```
LINE   12: }
LINE   13:
LINE   14: printd(n)
LINE   15: int n;
LINE   16: {
LINE   17:     register    i;
LINE   18:
LINE   9:      if (n < 0)  {
LINE   20:          putchar('-');
LINE   21:          --n;
LINE   22:     }
LINE   23:     i = 0;
LINE   24:     do  {
LINE   25:          s[i++] = n % 10 + '0';
LINE   26:     }  while ( (n /= 10) > 0 );

* break  26
```

(Sets another breakpoint at line 26.)

```
NEW BREAKPOINT SET AT: PROCEDURE  printd
LINE   26:     }  while ( (n /= 10) > 0 );

* go
```

(Continues the program.)

```
INFORM(35): EXECUTION SUSPENDED DUE
TO BREAKPOINT SET AT
printd:LINE   26: }  while ( (n /= 10) > 0 );

* ,n; ,s/s
```

(Displays the variables n and s.)

```
12345
5
```

```
* alias gd "go; ,n; ,s/s"
```

(Sets alias definition **gd** to resume execution and displays variables **n** and **s** in one command when the execution stops.)

```
* show alias
```

(Displays the alias definition.)

```
ALIAS: gd  DEFINED AS go; ,n; ,s/s
```

```
* gd
```

(Runs the command.)

```
INFORM(35): EXECUTION SUSPENDED DUE
TO BREAKPOINT SET AT
printd:LINE   26: }  while ( (n /= 10) > 0 );
1234
54
```

```
* gd
```

(Runs the command again.)

```
INFORM(35): EXECUTION SUSPENDED DUE
TO BREAKPOINT SET AT
printd:LINE   26: }  while ( (n /= 10) > 0 );
123
543
```

```
* delete  26
```

(Deletes breakpoint at line 26.)

```
* show break
```

(Displays the breakpoints.)

```
PROCEDURE  printit
LINE   34:          putchar(s[k]);
PROCEDURE  printd
LINE   19:     if (n < 0)  {
BKPT COMMAND:,n
PROCEDURE  main
LINE   11:     printd(i);
```

```
* delete all
```

(Deletes all the breakpoints.)

```
* go
```

(Continues the program. Since all the breakpoints are deleted, it will finish.)

```
12345
-987654

SYSTEM (99):  PROCESS TERMINATED
```

(Process is terminated whenever the main procedure completes execution.)

```
* quit
```

(Leaves the debugger program and returns the shell prompt.)

```
%
```

Now rerun the debugger program as follows:

```
$fdb a.out

Fortune Symbolic Debugger

* break 7
```

(Sets a breakpoint at line 7.)

```
NEW BREAKPOINT SET AT: PROCEDURE main
LINE 7:  i=12345;

* restart
```

(Restarts the debugger program.)

```
INFORM(35): EXECUTION SUSPENDED DUE
TO BREAKPOINT SET AT
main:LINE   7:            i= 12345;

* walk
```

(Single steps through the program.)

```
main:LINE   8:         printd(i);

* dump &i
```

(Displays the addres of i and the memory contents around the address of i.)

```
ADDRESS FOR :i IS 0x7ffed8
sp address  0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
-- ------- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
D  7ffdf0  00 00 00 00 00 00 00 62 00 00 30 39 00 00 00 00
```

*** !date**

(Executes the shell command for date and time.)

```
Fri Jul  8 12:57:23 GMT-3:30  1983
```

*** show file**

(Names the input, output, debug, and source files.)

```
INPUT FILE:  standard
OUTPUT FILE: standard
DEBUG FILE:  sample.c
SOURCE FILE: sample.c
```

*** find "print" 1**

(Searches for the first time the string **print** occurs from line 1.)

```
LINE    8:    printd(i);
```

*** find**

(Searches for the same string again.)

```
LINE   11:    printd(i);
```

*** print 1/10**

(Prints lines 1 - 10.)

```
LINE    1:  #defineMAXCHAR   10
LINE    2:  char   s[MAXCHAR];
LINE    3:  main()
LINE    4:  {
LINE    5:    int  i;
LINE    6:
LINE    7:    i = 12345;
LINE    8:    printd(i);
LINE    9:    i = -987654;
LINE   10:
```

*** show fun**

(Shows how the function keys are currently set.)

```
FUN KEY:       fun16 DEFINED AS  next
FUN KEY:        down DEFINED AS print next
FUN KEY:          up DEFINED AS print .-1
FUN KEY:     EXECUTE DEFINED AS walk in
FUN KEY:     execute DEFINED AS walk
FUN KEY:    nextscrn DEFINED AS print .-1!20
FUN KEY:    prevscrn DEFINED AS print .!-20
FUN KEY:        help DEFINED AS help
```

Now use the PREV SCRN and NEXT SCRN keys to see how they work as special function keys. The up arrow ( ) and the down arrow ( ) will print the line of source code above or below the current line. Type **q** to leave **fdb**.

# *Chapter 5*
# *Profiling a C Program*

The profiler is a tool designed to help programmers identify those parts of a program that are inefficient. It generates a list that indicates the amount of time spent executing each part of a program. This chapter describes how to use the profiler. It should be noted that the profiler alters the value returned by a program. In normal cases this should not cause any difficulty.

## COMPILING A PROGRAM FOR PROFILING

To prepare a file for profiling, you must first compile it with the **-p** option:

```
$ cc -p filename.c
```

The input to the **cc -p** command may contain no more than 200 procedures (the main program and all functions and subroutines).

The output of the **cc -p** command is a modified version of the executable file. The name of the executable file, as usual, is **a.out** unless the **-o** option was used and another name was specified.

The executable file produced by the **-p** option is modified to contain code that calls a program named **monitor. monitor** tracks the use of computer time during program execution.

## USING THE **prof** PROGRAM

Once the **cc -p** command line has prepared a file for profiling, the file must be executed as usual by typing its name and a return.

At the end of the execution, profiler information is placed in a file called **mon.out.** You may use the **ls** command to ascertain that **mon.out** is in your directory.

Once the a.out file has been executed, use the **prof** program to generate the profiler information. The basic syntax of **prof** is

```
$ prof
```

where your executable file is **a.out.** If **a.out** is not the name of your executable file, you must state the filename after the **prof** command:

```
$ prof filename
```

A listing of profiler information will appear on your screen in response to the **prof** command.

## A SAMPLE PROGRAM FOR USING THE PROFILER

This sample session illustrates the use of the profiler. The program to be profiled is named **proftst.c**. Its contents are

```
main()

{

     long i;
     for (i=0; i < 100; i++)
     fun();
     printf("Done\n");
}

fun()
{
     int n;
     for(n=0; n < 100; ++n)
          fun2();
}

fun2()
{
     int n;
     for(n=0; n < 10; ++n);
}
```

To compile the program, type

```
$ cc -p proftst.c
```

To generate the profiler listing, type

```
$ a.out
```

then the profiler command

```
$ prof
```

The following information will appear on your screen:

| %time | tm spent | cum sec | call | ms/call | name |
|---|---|---|---|---|---|
| 63.1 | 1.77 | 1.77 | 10000 | 0.18 | fun2 |
| 4.2 | 0.12 | 1.88 | 100 | 1.19 | fun |
| 0.0 | 0.00 | 1.88 | 1 | 0.00 | main |

where:

| | |
|---|---|
| %time | Percentage of time spent in function |
| tm spent | Time (in seconds) spent in function |
| cum sec | Cumulative time prior to this function |
| #call | Number of times function was called |
| ms/call | Number of milliseconds spent in call |
| name | Program unit name |

**NOTE:** The actual values displayed depend on the operating system, the C compiler, and Language Development Tools you are using.

# PART TWO
# THE C LANGUAGE

Part 2 of the Fortune C Language Guide summarizes the vocabulary and conventions of the C language on the FOR:PRO operating system. It defines

- The C character set and conventions of notation
- Data types and declarations
- Expressions and statements
- Routines
- Library functions
- Input and output

For a more comprehensive description of the C language, see Kernighan and Ritchie, The C Programming Language or one of the other books in the list of references in the introductory section of the guide.

# *Chapter 6*
# *The C Character Set and Conventions of Notation*

The C language uses a few conventions of notation that must be observed when programs are written. These as described, together with the C character set, in this chapter.

## THE CHARACTER SET

The C character set contains the following characters:

---

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

   0   1   2   3   4   5   6   7   8   9

     ! " ' # $ % & * ( ) + - _ / . , : ; < > =

             ? @ [ ] { }  ^  ¦ ~
```

---

The C language distinguishes between uppercase and lowercase. For example, the variable named "Lower," spelled with an uppercase L is different from the variable "lower," spelled with a lowercase l as the first letter. The same is true of constants.

## RESERVED IDENTIFIERS

In C, the following identifiers are reserved for use as keywords and cannot be used to identify other elements of a C program.

| | | |
|---|---|---|
| **auto** | **entry** | **return** |
| **break** | **extern** | **short** |
| **case** | **float** | **sizeof** |
| **char** | **for** | **static** |
| **continue** | **goto** | **struct** |
| **default** | **if** | **switch** |
| **do** | **int** | **typedef** |
| **double** | **long** | **union** |
| **else** | **register** | **unsigned** |
| | | **while** |

## NONGRAPHIC CHARACTERS

In the C language, the backslash \ is used with other characters to represent certain nongraphic characters as well as the single quote and the backslash itself. These non-graphic characters and their representations are as follows:

| | |
|---|---|
| backslash | \\ |
| backspace | \b |
| bit pattern | \ddd |
| carriage return | \r |
| form feed | \f |
| horizontal tab | \t |
| newline | \n |
| single quote | \' |

## STATEMENT FORMAT

The C language uses a free field for statements: They may begin or end in any column, and blanks beyond the first one are ignored by the compiler. Any number of blanks, tabs, or blank lines may be included to improve ease of reading.

## COMMENTS

The C compiler ignores any characters that appear between /* and */. These symbols can thus be used to create comments, that will appear in the program listing but not affect its execution. Such comments may be used freely in a C program. Comments can be included anywhere a blank or a newline can.

# *Chapter 7*
# *Data Types and Declarations*

The basic data types used by a C program are constants and variables. Constants are declared implicitly by their appearance and use in a program. Variables must be declared in such a way that the variable name becomes associated with a variable type. This type implies the properties and storage size of the variable.

The basic data types may also be used to create extended data types, such as structures and arrays. These extended data constructions are useful for modeling logical groups that arise in actual data.

Finally, pointer variables may be declared to point to both basic and extended data types. When used in place of array subscripts, pointers increase the efficiency of programs that perform index referencing.

The five classes of data types are described below.

## CONSTANTS

Constants may be characters, integers, or floating-point numbers. Characters may exist alone or in groups called strings.

### Character Constants

The following are examples of character constants:

```
'a' '1'      Single charaters
"January"    String of 8 characters (null included)
```

Character constants are enclosed by single quotes while strings are enclosed by double quotes. A string will always include the null character (a binary zero) as its terminator. This character is added by the compiler. Note that a string is the same as an array of characters in C.

### Integer Constants

Integer constants may be represented in different radix formats by prefixing them with the appropriate symbol, **0** for octal or **0x** for hexadecimal.

Here are some examples of integer constants:

```
-25 0 100000    Decimal constants

0177 011        Octal constants

0xa2 0xfff      Hexadecimal constants
```

### Floating-Point Constants

Floating-point constants are represented by "number decimal point number" or by scientific notation. They can also be expressed in hexidecimal or octal. The two ways of expressing floating-point constants are shown below.

```
12.5      1000.0001      Use of decimal points

23.4e23   10.023E-4      Use of scientific notation
```

## VARIABLES

A variable declaration consists of a variable name and an associated variable type. The variable name is a string of letters and digits, the first of which must be a letter.

### The Six Variable Types

There are six variable types in the C programming language. Each type implies rules for the manipulation of its value and the number of bits allocated for its storage, as shown in Table 7-1.

The additional **unsigned** qualifier may precede a character, short, or integer variable name to permit the integer to range in value from zero to a maximum of two to the n-minus-1 power, where n is the number of bits in the integer.

Multiple variables of the same type may be declared in a single declaration statement by separating each with a comma. Each declaration is terminated by a semicolon.

The following example shows the declaration of multiple variables of type **short**.

```
short   s1,s2,s3;
```

Table 7-1. C Data Types

| Type | Description | Storage in bytes | Example |
|---|---|---|---|
| **char** | character | 1 | 'a', '1', 'q' |
| **short** | integer | 2 | 32767, -53 |
| **int** | integer | 4 | 7494312 |
| **long** | integer | 4 | 9043555, -213328 |
| **float** | single-precision floating-point | 4 | 1E37, 3.976 |
| **double** | double-precision floating-point | 8 | 1E314, -E313 |

Precision Arithmetic

The program below illustrates precision arithmetic. Following the program is a table which gives the upper and lower bounds of all the numerical data types. C's precision is the same as Fortran's and follows the IEEE standard. In this standard, all real and double-precision numbers are represented in the form called normalized floating point form: plus or minus 2 raised to the n-th power times f where f is a real number whose lower bound may equal 1.0 and whose upper bound is always less than 2.0.

If you compile the program below by typing

```
$ cc program name -lm
```

and then run it by typing

```
a.out
```

you will get the following output:

```
uf is 3.4025878e+38
lf is 1.1754944e-38
ud is 1.797568532587847e+308
ld is 2.225073858507200e-308
little is 32767
big is 2147483647
```

The program statements beginning uf and lf (upper floating and lower floating) illustrate the largest and smallest floating point numbers expressed in normalized form. In the output above, they are converted to normal scientific notation. The double numbers ud and ld are treated similarly. The last two lines of the output show the largest short and the largest integer. (The function **pow** is described in section 3 of the FOR:PRO Programmer's Manual along with all the other C library functions.)

```
#include <math.h>
  /* this loads in the math library routines */
main()
{
short little; int i,big;
float uf,lf; double ud,ld;

uf =  pow( 2.0, 127.9999); printf(" uf is %1.7e 0,uf);
lf = pow( 2.0, -126.0); printf(" lf is %1.7e 0,lf);
ud =  pow( 2.0, 1023.9999); printf(" ud is %1.15e 0,ud);
ld = pow( 2.0, -1022.0); printf(" ld is %1.15e 0,ld);
little = 32767; big = 2147483647;
printf("little   is %5d0,little); printf("big  is %10d0,big);
}
```

Upper and Lower Bounds for Numerical Datatypes

| DATA TYPE | LOWER BOUND | UPPER BOUND |
|---|---|---|
| short | - | +(-)32767 |
| int | - | +(-)2147483647 |
| float | +(-)1.175494E-38 | +(-)3.402587E+38 |
| double | +(-)2.2250738507E-308 | +(-)1.7976931348E+308 |

The ffp Option

Users who want faster arithmetic operations and are willing to sacrifice precision and range can use the **-ffp** option with the C compiler. The necessary library, which comes with the Language Development Tools, is loaded automatically. Note that when you invoke **ffp**:

- There is no distinction between float and double, doubles are treated as floats.
- The range of the float exponent is reduced by approximately fifty percent.

The ranges for the **-ffp** option are:

```
-9.22337E18 < x < -2.71050E-20     (for negative values)
5.421010E-20 < x < 9.22337E18      (for positive values)
```

### A Fortune Extension: Long Variable Names

In many implementations of the C language, only the first eight characters of a variable name are significant. The Fortune C language, however, permits variable names to be of an arbitrary length.

The only limit imposed is one of practicality. Variable names that extend beyond 256 or even 128 characters will be unwieldy and a nuisance to type and understand. The underscore character may be useful in separating logical parts of a variable name.

This example illustrates a long variable name:

```
int global_user_directory_index;
```

## Mixing Variable Types (Casting)

A type is associated with each declared variable in C. When variables are used in expressions (such as relationals or assignments) these types are compared for compatibility. Normally, programs are written in such a manner that compatibility is obvious: Integers are added only to other integers, characters are compared only with other characters, and so forth.

When types do not match, implicit rules are provided, and the intended operation is able to continue. These rules are called casts. If a variable of type **short** is being compared or assigned to one of type **long**, for instance, the C-language cast converts the short variable to a long variable before it is compared or assigned.

In the following example, **s** is first moved to a long temporary variable or register, and its sign is extended, if necessary, to reflect a negative value. It is then ready for the assignment operation:

```
long i;
short s;

i = s;
```

In some cases the rules for casting are not defined; an explicit cast must be provided by the programmer.

Here is an example of an explicit cast:

```
char *cp;
short *sp;

sp = (short *)cp;
```

The type supplied in parentheses is the type to which the variable immediately following it should be converted. This may or may not generate additional machine instructions, but if the variables have been properly cast, the compiler will omit its warning.

Note that casting character pointers to **short** or **integer** may cause your program to fail due to an odd-byte address.

### A Fortune Extension: The Void Type

---

A special new void type, corresponding to the undefined type, is available with the Fortune C compiler. It may be used to prevent the "incompatible type" warning messages produced by the type-checking program **lint.**

In the following example, the function **func1** normally returns a long value. When the function is called and cast using the void type, a dummy variable that holds the return value is unnecessary because this return value is not needed.

This program illustrates the use of the void type:

```
main()

{
     (void)func1():
}

long
func1()

{
     int longval = 10;
     return(longval);
}
```

---

## ARRAYS

Array declarations permit the logical grouping of homogeneous data objects. Each data object occupies the same amount of physical space and is referenced by subscripts according to its position relative to the first element.

As in other languages, subscripts are used both in the declaration to indicate the size of the array and in the reference to indicate the element required. Arrays in C may have one or more dimensions.

### Simple Arrays

The simplest and most common array has only one dimension.

The following are examples of the declaration and indexing of one-dimensional arrays:

```
char cstring[80];    Array of 80 characters, termed a "string"
int iarray[100];     Array of 100 integers
iarray[0]            First integer in iarray
iarray[4]            Fifth integer in iarray
iarray[99]           Last integer in iarray
```

Since the first array element is indexed, the last element of an array is one less than the declared size of the array.

### Multi-Dimensional Arrays

Arrays are not limited to one dimension; they may have several. A two-dimensional array of integers is declared as follows:

```
int iarray[10][20];
```

A total of 200 integer elements are in this array. The elements are stored in order by rows. The rightmost subscript of the array varies fastest when the elements are accessed in the order in which they are stored in memory.

An array may be initialized by a list of initializing constants enclosed in braces. A multi-dimensional array is initialized by enclosing sublists of these constant initializers in braces.

Each of the sublists corresponds to an array row. In the event that the number of constants in a sublist does not match the number of elements in an array row, the elements of the array go uninitialized when there are too few constants or the constants are discarded when there are too many constants.

The following example shows a corresponding list and array:

```
int february[4][7] = {
        { 8, 8, 8, 8, 8, 0, 0 },
        { 8, 8, 8, 8, 8, 8, 0 },
        { 8, 8, 8, 8, 8, 8, 8 },
        { 8, 8, 8, 8, 8, 0, 0 }
};
```

## STRUCTURES

In C, structures combine heterogeneous data types into a compound data types. A C structure is equivalent to what is termed a "record", in many other languages. This compound data types collects data that logically belong together.

Structures may consist of combinations of simple variables, arrays, or additional nested structures. Structures of the same type may be grouped together into an array of structures.

Because **structure** is a declared type, pointers may point to structures to access individual elements. Combinations are possible, and structures may even contain pointers referencing themselves.

More information on passing structures and on pointers to structures appears in Chapter 10, "Library Functions."

### Simple Structures

Each element of a structure is a declared variable and may include both arrays and structures. This declaration has the following form:

```
struct [structure_tag] {
        element1;
        element2;
        element3;
        ...
        ...
        elementN;
} structure_name;
```

The keyword **struct** denotes the beginning of a structure declaration. The structure_tag notation merely names (or tags) the particular structure being declared and is therefore optional. However, such name tags should ordinarily be included because they are useful in the creation of additional instances of the structure and in the definition of pointers to the structure.

The following structure has one element that is a simple integer plus three elements that are character arrays:

```
struct tag_car {

     int   year;
     char  make[16];
     char  model[16];
     char  color[16];

} newcar;
```

Additional structures of this type (one integer and three character arrays) may be created using the same tag, **tag_car**:

```
struct     tag_car     usedcar;
struct     tag_car     junkcar;
```

Each of the above structures contains the same four elements as the original structure and is initialized by following the declaration with a list of constants enclosed in braces.

```
struct tag_car junkcar = {
                          1951,
                          "Buick",
                          "Super",
                          "black",
                         };
```

An element of a structure is referenced using the . operator:

```
junkcar.year          equals   1951
junkcar.make          equals   "Buick"
junkcar.model         equals   "Super"
junkcar.color         equals   "black"
```

Arrays of Structures

Multiple instances of the same type of structure may be grouped together into an array.

This example illustrates an array of structures:

```
struct person {

        int     age;
        char    name[32];
        struct  birthday {
                int     day;
                int     month;
                int     year;
        } bday;
        char    invention[32];

};

struct          person grade1[35];
struct          person grade2[32];
struct          person grade3[40];
struct          person *personptr;
```

The braces around the embedded structure birthday are optional.

The first structure definition does not occupy any space. It merely describes the elements of the structure. The tag **person** is used to reference this definition when declarations are created. Three arrays of structures were declared together with a pointer to a person structure.

To initialize an array of structures, enclose the initial values in braces and place them after the declaration. The values for each structure must also be enclosed in braces, as in the following example:

```
struct personinventors[35] = {
        {  33,
           "Barbara Wire", {25, 3, 50},
           "Barbed Wire"
        },
        {  32,
           "Walter Windchill", {14, 3, 51},
           "The Windchill Factor"
        },
        ...
    };
```

## Bit Fields

Structures may have special elements called bitfields. These bitfields are portions of integers used to replace masks that use bitwise operators.

The following format is used for bitfields:

```
struct {

     unsigned  fname1 : bitwidth1;
     unsigned  fname2 : bitwidth2;
     ...
     ...
     unsigned  fnameN : bitwidthN;

} struct_name;
```

Rather than using complex **#define** statements and bitwise operators, you can sometimes use bitfields to clarify the logic of the program.

Here is an example of complex programming:

```
#define  ORDERED  1
#define  SHIPPED  2
#define  PAIDFOR  4

int  flags;

flags |= ORDERED;

if((flags & (ORDERED | SHIPPED)) != 0) {
     ...
     ...
     ...
}
```

The above example can be replaced by this straightforward bitfield logic:

```
#define NO   0
#define  YES  1

struct {

          unsigned  is_ordered  :  1;
          unsigned  is_shipped  :  1;
          unsigned  is_paidfor  :  1;

} flags;
```

```
flags.is_ordered = YES;

if(flags.is_ordered == YES || flags.is_shipped == YES) {
        ...
        ...
        ...
}
```

Bitfields may not cross integer boundaries. If a sequence of bitfields is declared such that one would cross an integer boundary, the bitfield is aligned on the next full integer boundary.

Bitfields are not arrays and thus do not have addresses. They are declared as unsigned variables and are stored as integer variables. Attempting to retrieve the address of a bitfield by use of the & operator generates an error message.

## Unions

Unions are variants of structures that allow the user to overlay different types of data on the same allocated space. Rather than being allocated one after another, each element of a union uses the beginning of the union as its starting address. The compiler manages union size and union alignment requirements.

Unions, like structures, consist of one or more elements enclosed by braces; they follow the keyword **union**. The union_tag notation merely names (or tags) the particular union being declared and is therefore optional. However, such name tags should ordinarily be included because they are useful in the creation of additional instances of the union and in the definition of pointers to the union.

An example of the union and name tags follows:

```
union [union_tag] {
     element1;
     element2;
     ...
     ...
     elementN;
} union_name;
```

Unlike the structure variable, each of the elements of a union begins at the same location in memory. Unions may occur within structures and arrays, and vice versa. The structure operators . and -> are used to access elements of a union, together with the name of the union and a pointer to the union.

For example:

```
union_nametag.element2
```

The following union permits one of three types to be stored in the allocated space. The compiler reserves enough space to hold the largest-sized variable.

```
union u_3types {

     int    intval;
     long   longval;
     float  floatval;

} ilf_type;
```

POINTER VARIABLES

Pointer variables point to data objects by containing their memory addresses. They are declared like simple variables, but their names are preceded by the unary * operator.

The type associated with each pointer is the type of variable to which the pointer points. All pointers use the same amount of physical storage (4 bytes).

The following are examples of pointers:

| | |
|---|---|
| **int *pi;** | The pointer variable **pi** points to an integer |
| **int *pai[10];** | The array **pai** contains ten integer pointers |

A Fortune Extension: Non-Unique Structure Elements

Previously, all declarations at the same context level were required to be unique. The only exceptions to this rule were structure element declarations having the same physical offset within the structure.

These are examples of structure element declarations:

```
struct s_1{struct s_2  {
     int str_one;     int str_two;
     int same;        int same;
} s1;             } s2;
```

In these examples, although the integer **same** is non-unique in both structures, its offset from the beginning of the structure (its value) is the same. This has normally been permitted.

The following structure declarations each contain common elements that do not share the same value:

```
struct s_1 {struct s_2  {
     int same;       int str_two;
     int str_one;    int same;
} s1; }         s2;
```

The Fortune C compiler accepts these structural declarations by permitting non-unique structure element declarations that do not share the same offset value.

---

A pointer variable may be given a value by assigning an address to it. The address operator & (ampersand) is used to obtain an address.

Use the pointer variable in this way:

| | |
|---|---|
| **pi = &i;** | Assigns to **pi** the address of the integer i. |
| **pai[0] = &i;** | Assigns to the first element of the array **pai** the address of the integer i. |

The * operator is also used to obtain the value of the object to which the pointer points. This process is called de-referencing.

The following are examples of de-referencing:

```
int  i,j;
int  *pi;
```

| | |
|---|---|
| **pi = &i;** | Assigns to the pointer **pi** the address of **i** |
| **j = *pi;** | Assigns to **j** the value of the integer pointed to by **pi** |
| **j = i;** | Assigns the value of **i** to **j** |

The first two assignments above are equivalent to the third. In this example, the assignment **j = i** is more direct and efficient. However, had the situation included an array of integers within a loop, the de-referencing approach might have been more efficient. Here is an example of an array of integers within a loop:

```
int total = 0,iarray[10],*pi;

pi = &iarray[0];            Points to the first element of the
                            array iarray

while(pi =< &iarray[9]){    Means "while the pointer is not
                            pointing at array elements iarray
                            [0], through iarray[9]."

total += *pi;               Adds the element to the total

pi++;                       Advances the pointer to the next
                            element of the array iarray
}
```

Address Arithmetic

Addresses, like scalars, may be used in arithmetic expressions. Incrementing a pointer by one points at the next element of the storage. Adding a number to the pointer moves it ahead by the number of elements added.

Examples of incrementing a pointer follow:

```
long larray[10],*lp;

lp = larray;      Same as &larray[0]

lp++;             Moves the pointer to the next element

lp += 4;          Moves the pointer ahead 4 elements
```

You can omit the address of the first element in an array. Stating the name of the array is the same as stating the address of the first element. Therefore, the following two assignments are equivalent:

```
pi = &iarray[0];

pi = iarray;
```

Character Pointers

Character pointers (**char *name;**) may point at character arrays in addition to pointing at individual character variables. Character arrays may be string constants.

This is an example of a character array:

```
"This is a string constant"
```

This string constant is enclosed by double quotes and includes a binary zero \0 (supplied by the C compiler) as a terminator. String constants appear frequently in **printf** statements. The **printf** and other I/O functions in C are described in Chapter 11. If the address of the string constant is passed to the **printf** function the string constant will be printed. The following two calls to **printf** produce the same message.

This example illustrates the use of string constants:

```
printf("Hello universe\n");

char *message;
    message = "Hello universe\n";
    printf(message);
```

## Pointers to Structures

Pointers may also be declared to point at structures. As with simple variables, the asterisk * precedes the pointer name as follows:

```
struct    tag_date {
        intday;
        intmonth;
        intyear;
} date;

struct     tag_date *datep;
```

Once the pointer has been assigned the address of the structure, each element is referenced using the -> operator.

This is how the elements are referenced:

```
datep =  &date;          Points the pointer at the structure

          datep->day     References each of the elements
          datep->month
          datep->year
```

## Pointer Arrays

Pointer arrays extend the usefulness of pointers by permitting the association of a group of elements, such as string constants of variable length. The following declaration creates a look-up table for the days of the week.

This example illustrates the pointer array:

```
char *days[] = {
     "Nobodaddyday",
     "Monday",
     "Tuesday",
     "Wednesday",
     "Thursday",
     "Friday",
     "Saturday",
     "Sunday"
};
```

A dummy string "Nobodaddyday" has been inserted as element zero in order to force array numbers to match the conventional one-through-seven numbering of weekdays. The numeral 8 is not required inside the brackets of the declaration because the C compiler counts the subsequent list and generates the correct number.

## Pointers Used as Function Arguments

Pointers may be passed as arguments to functions. When so passed, the variable to which the pointer is pointing is being passed "by reference."

Arguments are ordinarily passed to functions "by value" (that is, a copy is made and handed to the function). When arguments are passed by value, changes made to the copy have no effect on the original. However, both the copy and the original may be changed when passing is by reference. In this example, the variable i is being passed by value:

```
main()
{
     int i = 25;
     func(i);
     printf("The value is still %d\n",i);
}

func(i)
int i;
{
     i = 0;
}
```

Although the integer **i** is passed to the function **func** and set equal to zero, it continues to have the value 25 in the main function. This is not the case in the call-by-reference example below. The value is changed, by in executing the function.

```
main()
{
     int i = 25,*pi;

     pi = &i;

     func(pi);

     printf("The value is now %d\n",i);
}

func(ppi)
int *ppi;
{
     *ppi = 0;
}
```

In addition to passing arguments by reference for the purpose of modifying the original, pointers are passed to functions as a way to avoid passing an entire array or structure. This use of pointers along with address arithmetic permits access to each element or field of the larger data objects.

This example passes pointers to functions:

```
main()
{
     int iarray[20],*pi;

     pi = &iarray[0];

     func(pi);
}

func(ppi)
int *ppi;
{
     int i;

     i = *(ppi + 3);    'i' becomes equal to element
                             iarray[3] of the array
}
```

# *Chapter 8*
# *Expressions and Statements*

The C language provides a large set of operators for creating expressions. It uses control-flow statements to determine the order in which the actions of a program are executed. The types of operators and statements and their uses are described below along with examples of their use.

OPERATORS

The C operators can be grouped under seven major headings:

1. Arithmetic Operators

| | |
|---|---|
| - | Unary minus |
| + - | Binary add, subtract |
| * / % | Binary multiply, divide, modulus |

2. Logical Operators

| | |
|---|---|
| ~ | Unary one's complement |
| & | Bitwise AND |
| \| ^ | Bitwise inclusive OR, exclusive OR |
| << >> | Left shift, right shift |
| && \|\| | Binary AND, binary OR |

3. Relational Operators

| | |
|---|---|
| == != | Equal and not equal |
| < <= > >= | Less than, less than or equal, greater than, greater than or equal |

4. Assignment Operators

| | |
|---|---|
| = | Assignment |
| += -= *= /= %= | Assignment after arithmetic operation |
| <<= >>= &= ^= \|= | Assignment after logical operation |

5. Conditional Operator

| | |
|---|---|
| ?: | Ternary conditional |

6. Increment/Decrement Operators

| | |
|---|---|
| ++ -- | Increment by one, decrement by one |

7. Special Operators

| | |
|---|---|
| () | Change order of evaluation |
| [] | Array reference |
| -> | Pointer to a structure member |
| . | Structure or union element |
| (type) | Type cast |
| * | Pointer reference |
| & | Variable address |
| **sizeof** | Object size in bytes |
| , | Multiple expression |

Order of Operations

Table 8-1, reprinted from page 49 of Kernighan & Ritchie,* summarizes the special operators. Rows are in order of decreasing precedence.

Table 8-1. Order of Operations

| Table of Operators | Associativity |
|---|---|
| () [] -> . | Left to right |
| ~ ++ -- - (type)<br>* & sizeof | Right to left |
| * / % | Left to right |
| + - | Left to right |
| << >> | Left to right |
| < <= > => | Left to right |
| == != | Left to right |
| & | Left to right |
| ^ | Left to right |
| \| | Left to right |
| && | Left to right |
| \|\| | Left to right |
| ?: | Right to left |
| = += -= *= /= %=<br><<= >>= &= ^= \|= | Right to left |
| , (used in the **for** statement) | Left to right |

*From Kernighan, Brian W. and Ritchie, Dennis M.: The C Programming Language, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

## EXPRESSIONS

Operators are combined with variables and constants of different types to form expressions. Expressions can be nested. They are evaluated according to the rules of precedence of the operators. The evaluation of an expression has two effects: The variables that make up the expression may be changed, and the value of the expression may be used to determine an action in a program.

## STATEMENTS

Control-flow statements determine the order in which the actions of a program are executed. Sometimes this order depends on the execution of an expression. Each statement should be chosen carefully to reflect the logical intent. In some unusual cases, it may be difficult to represent the required flow with a single control statement. Such situations can often be handled either by combining two or more control-flow statements or by using the single semicolon as a null statement.

The simpliest statements are expressions that are terminated by semicolons, as in the following examples:

```
i = j;
x--;
func(abc);
```

Left and right braces { and } are used to group declarations and statements into compound statements or blocks. One of these blocks is syntactically equivalent to a single statement. A terminating semicolon never follows the right brace of a block.

The following program illustrates this use of braces:

```
main()
     { int i = 0;   /* declaration in the block */

     while(i < 10)
     {
          int j; /* declaration in an inner block */

          j = i * i; i++;
          printf("j = %d\n",j);
     }
}
```

The control-flow statements in C are as follows:

### if-else: Choosing Between Two Execution Paths

The **if-else** statement permits a program to choose between two execution paths. It takes the form

```
if (expression)
     statement_1
else
     statement_2
```

The **else** portion is optional. The expression is evaluated, and if the result is true or non-zero, statement_1 is executed. Otherwise (if the expression is false or zero), statement_2 is executed. In the event that statement_1 is also an **if-else** statement, an ambiguity might arise regarding the connection of the trailing **else** statement:

```
if(expression_1)
     if(expression_2)
          statement_1
     else
          statement_2
```

The **else** portion of an **if-else** is always associated with the closest previous **if** statement without an **else**. In the above example, the **else** associates with the **if** using expression_2. The **if**, using expression_1, has no **else** statement at all.

If the **else** portion were required to associate with the **if** statement using **expression_1**, then braces would be needed to denote the completion of the inner **if** statement, as follows:

```
if (expression_1){
     if (expression_2)
          statement_1
}
else
     statement_2
```

A number of **if-else** statements may be connected together to create complex logic trees. In such a construct, the trailing **else** acts as the default condition when none of the previous expressions obtain. For example:

```
if (expression_1)
     statement_1;
else if (expression_2)
     statement_;
else if (expression_3)
     statement_3;
else if (expression_4)
     statement_4;
else
     statement_default;
```

## switch: Testing Against Constant Values

The **switch** statement permits an expression to be tested against a number of constant values. An action may then be taken, based upon a match.

Here is an example of the **switch** statement:

```
switch (expression){
     case constant1:
          statement_1;
          break;
     case constant2:
          statement_2;
          break;
     .
     .
     .
     case constantN:
          statement_N;
          break;
     default:
          statement_def;
          break;
}
```

Once the expression has been evaluated, it is compared with each of the constants in the case portions of the statement. Upon encountering the first match, the program proceeds to execute the statements following the colon. Execution continues until either a break statement or the enclosing brace is reached.

In the event that the expression does not match any of the constants, the optional default case is taken. Otherwise execution proceeds after the enclosing brace.

The C **switch** construct is similar to the case construct in Pascal. This program illustrates its use:

```
/*  Count the number of a's, b's, and c's in a string  */

main()
{
     int achars = 0;
     int bchars = 0;
     int cchars = 0;
     int others = 0;

     char *string = "Count my a's, b's, and c's!";
     char *p;

     p = string;
```

```
        while (*p != '\0'){
             switch (*p){
                  case 'a':
                       achars++;
                       break;
                  case 'b':
                       bchars++;
                       break;
                  case 'c':
                       cchars++;
                       break;
                  default:
                       others++;
                       break;
             }
             p++;
        }
        printf ("a's = %d, b's = %d, c's = %d, others = %d\n",
             achars, bchars, cchars, others);
    }
```

while: Repeating Until an Expression is False

The **while** statement permits an action to be repeated each time that a control expression evaluates non-zero or true. Once the expression evaluates to zero or false, execution continues with the next statement after the **while**. The expression is always evaluated before deciding to execute the statement. This is the format of the **while** statement:

```
while (expression)
     statement
```

The following program shows the use of the **while** statement:

```
main()
()
     int i = 1;
     char *string = "Look for the next blank space";
     char *p;

     p = string;
     while (*p != ' '){
          i++;
          p++;
     }
     printf ("The next blank was found at position %d\n",i);
}
```

Note that the increment operator can be included in the expression to reduce the number of statements. The following statement shortens the program:

```
while (*p++ != ' ')
        i++;
```

The only difference will be the value of the pointer **p** when the expression is not equal or false. The pointer **p** in the first example will point to the blank. In the second example, it will point to the character following the blank.

## for: Initializing and Incrementing

The **for** statement builds on the while statement by including the initialization and incrementation steps in the syntax. The **for** statement uses the following format:

```
for (expression_1; expression_2;
     expression_3) statement
```

Upon entering the **for** statement, expression_1 is executed once. It will not be executed again unless the **for** is entered from the top.

Next, expression_2 is evaluated and, if non-zero or true, the statement of the **for** will be executed. Once the statement has been executed, expression_3 is evaluated. If expression_2 is zero or false, execution continues after the statement. Usually expression_1 is an assignment while expression_2 is a relational expression. The **for** statement is functionally equivalent to the following set of statements:

```
{
expression_1;
while (expression_2){
     statement
     expression_3;
}
```

This program causes a loop:

```
int iarray[10] = { 1, 2, 4, 8, 16, 0, 32, 64, 128, 256 };

main()
{
     /* check each value of an array for zero */

     int i;

     for (i = 0; i < 10; i++)
          if (iarray[i] == 0)
               printf ("A zero found at %d\n",i);
}
```

Any of the three expressions in a **for** statement may be omitted, although the semicolons must remain. If all three expressions are omitted, the **for** degenerates into an infinite loop:

```
for (;;)
     statement
```

Presumably this endless loop could be broken by use of a **break**, **return**, or **goto** statement.

This example uses the **break**, **return**, and **goto** statements to exit the loop:

```
main()
{
     /* exit from an infinite loop on certain conditions */

     int i = 0;
     int j = 13;

     for (;;){
          j *= j;
          j -= i;
          if (j == 9999){
               printf ("Found the 9999\n");
               break;
          }
          if (j >= 9999){
               printf ("Out of bounds\n");
               return;
          }
```

```
            if ((j % 1000) == 0){
                printf ("Multiple of 1000\n");
                goto m1000;
            }
            i++;
        }
    m1000:
        printf ("A compound statement could go here\n");
    }
```

do-while: Repeating While an Expressions is True

The **do-while** statement permits a statement to be executed before the conditional expression is evaluated. This is the format of the **do-while** statement:

```
do
    statement
while (expression);
```

Upon entry, the statement is executed. Next, the expression is evaluated and if non-zero or true the loop repeats. If the expression is zero or false, execution continues with the next statement after the **do-while**.

The following program illustrates the **do-while** statement:

```
main()
{
    /* count the characters in the string */

    int  i = 0;
    char *string = "Here is a string of characters";
    char *p;

    p = string;
    do{
        i++;
    }while (*p++ != '\0');
    printf ("The string is %d characters in length\n",i);
}
```

break: Exiting a Loop

The **break** statement is used to exit a **for, while,** or **do** statement from a place other than the top or bottom of the statement. Execution continues with the statement immediately following the loop.

The following statement illustrates the use of the **break** statement:

```
for (i = 0; i < j; i++){
     ...
     ...
     if (iarray[i] < 0)
          break;
     ...
     ...
}

while (i < j){
     ...
     ...
     if (++i == 0)
          break;
     ...
     ...
}
```

Remember that the **break** statement is also used in the **switch** statement to direct control to the end of the **switch.**

continue: Starting the Next Iteration of a Loop

The **continue** statement is used within a **for, while,** or **do** statement to cause the next iteration of the loop to occur. In the **for** statement, the third expression is executed. In the **while** and **do** statements, the test expression in parentheses is executed immediately.

A typical use of the **continue** statement is the following:

```
for (i = 0; i < 100; i++){
     ...
     ...
     if (iarray[i] < 0)/* skip negative elements */
          continue;
     ...
     ...
}
```

Note that the **continue** statement applies only to loops and has no meaning in a **switch** other than to execute the next iteration of an enclosing loop.

## goto and Labels: Creating a Branch

The **goto** permits a branch to a defined label elsewhere in the same function. This label must be either in the same block or in an enclosing block. Normally, this statement is used to escape from a deeply nested series of statements upon encountering an error.

The **goto** statement can be used in the following way:

```
while(...){
        ...
        ...
        if (special_case)
             goto exit_label;
        ...
        ...
}
exit_label:
        ...
        /* exit code */
        ...
```

# *Chapter 9*
# *Routines*

Programs written in C usually consist of several routines in a single file. Data is passed to functions through agreements or through variables that have been declared in the program. The components of a function are described below:

## FUNCTION NAMES

Every function within a C program must have a unique name. This name is external to the function in the sense that it is available to all other functions in the program. Alphanumeric characters and the underscore _ may be used to create such names. The format is as follows:

```
func_1()
{
     ...
     ...
     ...
}
```

Every C program must contain a function with the name **main** because this is the runtime entry point at which execution of the program begins.

The following example shows the skeleton for a C program:

```
main()
{
     ...
     ...
}
```

## FUNCTION STRUCTURE

In addition to its name, a function may include the following components:

```
type                        optional
fname (argument_list)       optional
argument_declarations       optional
{
     declarations           optional
     statements             optional
}
```

As can be seen, many of a function's structures are optional. Every function returns a value of some type. Whether this value is used depends on the function's caller's declaration. If **type** is omitted, the return value's type defaults to **integer.**

Arguments may be passed to a function through the argument list. A declaration for all arguments that are not of type **integer** should appear in the argument-declaration list. It is good practice to include declarations for all arguments on this list. In the event that no argument is used, enclosing parentheses ( ) must still be used.

The body of the function is surrounded by enclosing braces { }. Optional declarations and program statements may be placed within the braces. The **return** statement, when present, specifies the return value upon **exit** from the function. A minimal function looks like:

```
minimal()
{
}
```

The skeleton of a function that converts an ASCII string of digits into an integer might look like:

```
int
coi(string,size)    /* ASCII to integer */
char *string;       /* pointer to chars */
int  size;          /* length of string */
{
     int intval;
     ...
     ...
     ...
     return(intval);
}
```

ARGUMENTS

Arguments are passed to functions through the argument list. Generally, each argument is passed by value; that is, a copy of the actual data item is given to the function. Any changes that the function makes to this copy will not affect the original value.

One exception to this rule arises when array names or pointers to data objects are passed as arguments. In this case, an address that points to the original data is passed, and modification of data objects will change the original data. This pass-by-reference technique is used when a function needs to return more than a single value.

When passing arguments, take care to assure that two essential conditions are satisfied. First, data objects passed as arguments to a function should match in type with the argument declaration of that function. Passing a long argument when a short argument is expected can lead to trouble.

Second, the argument list should contain the same number of data objects as the number passed in the function call. If the number of objects expected is greater than the number passed, the function will receive arbitrary values for some arguments.

A Fortune Extension: Passing Structures by Value

---

Older implementations of the C language permitted structures to be passed to functions "by reference" only. That is, rather than making a copy of the argument and pushing it onto the stack, a pointer was passed to reference the remainder of the variable.

The Fortune implementation of the C language permits structures to be passed by value as well as by reference. The following example passes both the address and a copy of date:

```
struct date_s{
     int month;
     int day;
     int year;
}date;

func(&date);  passes the address of "date"

func(date);  passes an entire copy of "date"
```

Each function, on return from its execution, may pass a value back to its caller. This returned value may in turn be used to build expressions.

---

## RETURN VALUES

Upon encountering a **return** statement, a function exits and returns a value available to the caller. This value may be used in an expression.

If the return value is a type other than **integer**, it must be declared ahead of the function name. This declaration alerts the return mechanism to save an adequate amount of temporary space.

Because the **return** statement syntax does not require a value, be careful to include one if the return value is used by the caller.

### A Fortune Extension: Structures as Return Values

Just as structures can be passed by value to functions, Fortune's version of C permits structures to be returned as the value of a function. By declaring a function to return a structure, sufficient space is reserved when the function is called to contain the entire return value. The example below declares function **getdate** to return the structure **date_s**:

```
struct date_s{
     int month;
     int day;
     int year;
}date;

struct date_s getdate();

main()
{
     struct date_s d;

     d = getdate();
}

struct date_s
getdate()
{
     struct date_s gdate;
     ...
     ...
     ...
     return(gdate);
}
```

## USING EXTERNAL, STATIC, AND REGISTER VARIABLES

Variables may be assigned to one of three storage classes in a function or program: **external**, **static**, or **register**. The assignment depends on how many functions use them and how often they are used.

## External Variables

A variable is external if it is declared outside of a function. Such variables are considered global in that they are available to other functions. All functions recognize these variables by name. If a program's functions are declared across two or more files, an **external** declaration might be required in the file that does not have the definition.

All external variables are by default initialized to the value zero and may be reset to other values by declaring them in the following way:

```
int counter = 100;
```

## Static Variables

Static variables are similar to external variables in some respects and similar to automatic variables in other respects. They are allocated like external variables and exist during the entire program execution. They differ from external variables in scope. When a static variable is declared with the external variables of a file, it is available only to the functions of that particular file, and when a static variable is declared within a function, it is available only to that particular function. Like the value of an external variable, the value of a static variable may be passed between calls to functions in the same file (that is, the file in which it is declared).

Like external variables, static variables have the value zero unless they are explicitly initialized to another value. The following example shows the static variable initialized to 1200.

```
/* external static */

static int estat_i = 1200;

main()
{    /* internal static */

     int i;
     static int estat_i;

     ...
     ...
     ...
}
```

## Register Variables

Register variable declarations should be substituted for any function arguments and local automatic variables accessed frequently by the program. Register variables are stored in processor registers rather than in temporary memory locations and are more efficiently accessed. The use of register variables may increase execution speed.

The Fortune C compiler supports a maximum of six data register variables plus a maximum of six floating-point register variables in addition to a maximum of six address register variables (such as those used to store pointers). If there are more register declarations than machine registers, the excess declarations are not lost but become normal arguments or automatic variables.

The following example shows the method of register declaration.

```
func(a)
register int a;
{
     register int i;
     register float f;
     ...
     ...
     ...
}
```

## INVOKING A FUNCTION

Functions are invoked by using their names and a pair of parentheses. Any arguments to be passed to the function must appear within the parentheses. Function invocations may stand alone as separate statements or they may be part of an expression.

Several ways to call a function are shown below:

```
function(arg1, arg2);
retval = function(arg1, arg2);
x = y + function(arg1, arg2) + z;
```

In the first example, the value returned from the function is not used. In the second, the return value is assigned to the variable retval. It is important that the type of the value returned from the function be the same type as the variable to which it is assigned. In the third example, the value returned from the function is used to form an expression.

Functions in C may call themselves recursively. Each invocation of the function receives a new set of automatic variables. Because it is sometimes difficult to predict the number of

recursions, the user must be alert to overflowing of the run-time stack.

The following example shows the function **printd** calling itself recursively.

```
printd(n)  /* print n in decimal */
int n;
{
     int  i;

     if (n < 0){
          putchar('-');
          n = -n;
     }
     if ((i = n/10) != 0)
          printd(i);
     putchar(n % 10 + '0');
}
```

## DESIGNING FUNCTIONS FOR EASE OF USE

Whenever possible, each C function should be designed to perform one simple set of operations. Large monolithic functions that attempt to do everything should be avoided. In general, each C function should consist of no more than a page or two of code. Small size makes each function easy to read and modify.

As programs grow larger, the number of functions in the C source file may increase. To avoid the extra time required to compile the entire source file for minor additions or corrections, divide the C functions into two or more source files. These files are compiled individually then are linked together to form the executable object module.

Commonly used functions may be placed in libraries for easy access using the archive program (**ar**), which creates and maintains library files. Since **loader** operates at the file level, it is best to limit each library file to a single function. Otherwise, when the library is searched and a function is found, all the other functions in that file will also be loaded, and the output will be unnecessarily large.

# *Chapter 10*
# *Library Functions*

The FOR:PRO operating system provides a number of libraries that can be called from C programs. They include

- Math functions
- String functions
- Memory management functions

This chapter describes these library functions. They are also discussed in Section 3 of the FOR:PRO Programmer's Manual.

## MATH FUNCTIONS

A library of sophisticated mathematical functions is available to the programmer for use in creating C programs. In all cases, the file **math.h** should be included in programs using the math library.

The format for using the math functions is

```
#include <math.h>
```

Each of the math functions is listed below with descriptions of the appropriate argument and return types.

### Exponential

```
double exp (x)      Returns the exponential Function of x
double x;
```

### Logarithmic

```
double log (x)      Returns the natural Logarithm of x
double x;
```

### Base ten logarithmic

```
double log10 (x)    Returns log to the Base 10
double x;
```

Power

**double pow (x,y)**
**double x, y;**

Returns x to the y power

Square root

**double sqrt (x)**
**double x;**

Returns the square root of x

Absolute value

**double fabs (x)**
**double x;**

Returns the absolute value of x

Floor

**double floor (x)**
**double x;**

Returns the largest integer - as a double precision number - not greater than x

Ceiling

**double ceil (x)**
**double x;**

Returns the smallest integer not less than x

Log gamma

**double gamma (x)**
**double x;**

The sign of gamma (x) is returned in the external integer signgam. The following C program fragment calculates gamma:

```
y = gamma (x);
if (y>88.0)
     error ();
y = exp (y) * signgam;
```

Euclidean distance

**double hypot (x,y)**
**double x,y;**

Returns square root(x*x + y*y), taking precaution against unwarranted overflows

Bessel functions

```
double j0 (x)
double x;
double j1 (x)
double x;
double jn (n, x)
double x;
double y0 (x)
double x;
double y1 (x)
double x;
double yn (n, x)
int n;
double x;
```

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

Trigonometric functions

```
double sin (x)
double x;
double cos (x)
double x;
double tan (x)
double x;
```

Sin, cos, and tan return trigonometric functions of radian arguments

```
double asin (x)
double x;
double acos (x)
double x;
double atan (x)
double x;
double atan2 (x,y)
double x, y;
```

Returns the arc sin in the range Pi/2 to Pi/2

Returns the arc cosine in the range to Pi

Returns the arc tangent of in the x range -Pi/2 to Pi/2.

Returns the arc tangent of y/x in the range -Pi to Pi.

Hyperbolic functions

```
sinh (x)
double x;
double cosh (x)
double x;
double tanh (x)
double x;
```

These functions compute the designated hyperbolic functions for real arguments.

## STRING FUNCTIONS

A library of the string functions available to C programmers is automatically searched whenever the loader pass of **cc** is invoked. Each of the string functions in this library file is listed below.

Note that the following functions will operate on null-terminated strings and that no check is made for overflow in the receiving string.

### String Concatenation

```
char  *strcat(s1, s2)
char  *s1, *s2;
```

Appends a copy of string s2 to the end of string **s1** returning a pointer to the null-terminated result.

```
char  *strncat(s1, s2, n)
char  *s1, *s2;
int   n;
```

Appends a copy of at most n characters of string **s2** to the end of string **s1**, returning a pointer to the null-terminated result.

### String Compare

```
strcmp(s1, s2)
char  *s1, *s2;
```

Compares string **s1** to string **s2** returning an integer greater than, equal to, or less than zero according to the lexicographical result of that comparison

```
strncmp(s1, s2, n);
char  *s1, *s2;
int   n;
```

Compares at most n characters of string **s1** to string **s2** returning an integer greater than, equal to, or less than zero according to the lexicographical result of comparison

### String Copy

```
char  *strcpy(s1, s2);

char  *s1, *s2;
```

Copies string **s2** to string **s1**, stopping after the first null character is copied and returning a pointer to the new string **s1**

```
char  *strncpy(s1, s2, n);   Copies n characters
                             of string s2
char  *s1, *s2;              to string s1, returning a pointer
int   n;                     to the new string s1
```

Note that the new string **s1** may not be null-terminated if the value of n was greater than the total length of string **s2**.

String Length

```
strlen(s);                   Returns the number of non-null
char *s;                     characters in string s
```

Index

```
char  *index(s, c);          Returns a pointer to the first
char  *s, c;                 occurrence of character 'c' in
                             string s or returns a zero if
                             that character cannot be found

char  *rindex(s,c)           Returns a pointer to the last
                             occurrence of character c in
                             string s or returns a zero
                             if that character cannot be found

char  *s, c;
```

## MEMORY MANAGEMENT FUNCTIONS

An executing program may dynamically allocate and free segments of main memory for use as data space. Two C library routines, **malloc** and **free**, provide this service.

### malloc: Allocates a Block of Contiguous Memory

The library routine **malloc** searches a circular list of available main memory space and allocates a block of contiguous memory locations. The format for the **malloc** routine is:

```
char *malloc(size)
unsigned size;
```

Following is an example of the use of **malloc**:

```
char *newspace;
unsigned mspace = 512;

newspace = malloc(mspace);
```

A pointer to the allocated memory space is returned from the call to **malloc**. This pointer must be used when the space is freed. A null pointer is returned if there is no available free memory or if the memory area has been corrupted.

free: Returns Allocated Memory to Pool of Available Memory

The **free** routine is used to return allocated main memory to the pool of available memory. The pointer returned from the **malloc** call must be used to free the proper memory. The free memory area may be corrupted if a random pointer is used.

The **free** routine format is

```
free(memptr)
char *memptr;
```

An example of the use of the **free** routine follows:

```
char *mptr;
int unsigned mspace = 1024;

mptr = malloc(mspace);
...
...
...
free(mptr);
```

It is not necessary to indicate the size of the segment of main memory to be freed. The FOR:PRO operating system keeps track of this space.

setbuf: Flushing I/O

Ordinarily, the buffered standard I/O library routines **fopen**, **fread**, and **fwrite** should be used when performing I/O. Buffered streams are provided to minimize the number of requests to the FOR:PRO kernel.

The **setbuf** call is used after a stream has been opened but before it is **read** or **written**. It forces a user-supplied buffer to be used in place of an automatically allocated buffer.

Use this format for the **setbuf** function:

**setbuf(**stream, buffer**);**

The stream parameter is the value returned from an **fopen** call. The **buffer** parameter is a pointer to a character buffer (possibly returned from a **malloc** request for space). If the constant pointer NULL is used in place of a buffer pointer, I/O through that stream will be completely unbuffered, making it equivalent to the functions **open**, **read**, and **write.**

setfree: Allocating Free Space

Normally, free space is managed through FOR:PRO by making calls to **malloc** and **free.** These calls permit free space to be allocated dynamically.

If a segment of space is known to be required before a process begins execution, it is more economical to make it available using the **setfree** system call rather than repeated **malloc** and **free** system calls. This is a feature specific to the Fortune implementation of UNIX.

The **setfree** call has the following syntax:

**setfree(**pointer, size**);**

The parameter pointer points to a block of memory to be allocated to the new free area. The parameter size indicates its size in bytes. For more information on storage management, refer to malloc(3) in the FOR:PRO Programmer's Manual.

# *Chapter 11*
# *Input and Output*

C does not include the I/O primitives for moving data between the processor and system peripherals. Instead, a number of input and output functions are available through a library file that is linked automatically whenever an executable object file is created using **cc**. This library is termed the **standard I/O library** and has been designed to be both efficient and portable.

The following sections describe many of the functions provided in this standard I/O library file. When using library routines, it is necessary to include a header file. The following statement should appear near the beginning of every source file that uses the I/O routines:

```
#include <stdio.h>
```

The **stdio.h** file contains definitions required by the standard I/O library.

A description of every library routine would be beyond the scope of this section. Consult Section 3 of the FOR:PRO Programmer's Manual for more information.

## STANDARD INPUT AND OUTPUT

Standard input and output are concepts designed to connect processes together easily. The default standard input device is the keyboard and the default standard output device is the display screen. At the simplest level, a program reads or writes a single character by calling a routine.

The following are examples of the format of input and output statements:

```
char c;
```

| | |
|---|---|
| **c = getchar();** | Reads a single character from the standard input. |
| **putchar(c);** | Writes a single character to the standard output. |

More general versions of these single-character input and output routines are used to construct more sophisticated routines. In addition, the **scanf** and **printf** functions can be used to read and print formatted input and output. These are used as described below.

## scanf: Reading Formatted Input

The function **scanf** reads formatted data from the standard input. This is the format of the **scanf** function:

**scanf(**format, variable1, variable2, ...**);**

This function reads from the standard input and assigns values to variables based on the format described in the first argument. Conversion specifications may be present in the format string to transform ASCII characters to numeric values. These specifications are preceded by a **%** and are followed by conversion characters such as **d** for decimal, **o** for octal, or **s** for string.

The following is an example of the use of the **scanf** function:

```
int year;
float model;
char company[32];

scanf("%d %f %s", &year, &model, company);
```

If you use this program with the input line:

```
1983    32.16Fortune
```

it will assign 1983 to year, 32.16 to model, and "Fortune\0" to company.

Note that since **scanf** expects arguments to be passed "by reference," you must supply addresses for arguments.

## printf: Directing Formatted Output

The **printf** function directs a string of characters to the standard output device. This string consists of an exact text string plus optional formatted variable values based on conversion characters in the text string.

The **printf** function format looks like this:

**printf(**format, variable1, variable2, ...**);**

As with **scanf**, the format string directs conversion of the variables that follow. ASCII characters are copied from the format string to the standard output device.

The **%** denotes a variable value conversion for the next variable in order.

The following program fragment illustrates the use of the **printf** function:

```
int year = 1983;
float amount = 98.27;
char *company = "Acme";


printf("In %d, the amount of %f was owed by %s.\n",
     year, amount, company);
```

This example prints:

```
In 1983, the amount of 98.27 was owed by Acme.
```

## ACCESSING FILES

Files not connected with a program through command line redirection may be opened and accessed using functions in the standard I/O library. The routines described below buffer data and call lower level FOR:PRO interface routines, which are described in Chapter 11.

### fopen: Opening a File

Before a file can be read or written, it must first be opened. The declaration and calling sequence to open a file are

```
FILE *fopen(), *streamp;

streamp = fopen(filename, mode);
```

The *filename* argument is a pointer to a string corresponding to the name of the file to be opened. The second argument, *mode*, specifies the open mode. Allowable modes are: read (**r**), write (**w**), and append (**a**), which writes at to the end of the file.

If a file is opened for either writing or appending and the file does not exist, it is dynamically created. If the file does exist and is opened for writing, the previous contents are erased, and the file is truncated to zero.

The value streamp returned from **fopen** is called a stream pointer. It points to an area of the data type **file** that is declared in **stdio.h** and buffers data between the file and the program.

## fread: Moving Data to the Program Area

The **fread** library function permits data to be moved from a file into the program area.

The format for using **fread** follows:

**bytes = fread(**ptr, sizeptr, numitems, streamp**);**

The first argument to the **fread** routine is a pointer or an address **ptr**, which points to the place where the data is to be transferred. In the above syntax, sizeptr is the size in bytes of the type of item to be transferred, numitems is the number of items of that type to be transferred, and streamp is the file pointer returned from an **fopen** call. The routine returns the number of items that are described by sizeptr and are actually read.

## fwrite: Moving Data from the Program to a File

The **fwrite** library function permits data to be moved from the program area into a file.

Use the following format for the **fwrite** function:

**bytes = fwrite(ptr, sizeptr, numitems, streamp);**

The first item to the **fwrite** routine is a pointer ptr that points to the place from which the data is to be transferred. Then sizeptr is the size in bytes of the type of item to be transferred; numitems is the number of items of that type to be transferred; and streamp is the file pointer returned from an **fopen** call. The routine returns the number of items actually written.

## fclose: Closing a File

The **fclose** library routine closes a file that has been previously opened by an **fopen** call. Buffered data is flushed and written to the file before it is closed.

The format of the **fclose** routine is

**fclose(**streamp**);**

The streamp argument uses a file pointer previously returned from an **fopen** call.

fscanf and fprintf: Using Stream Pointers

These routines behave like **scanf** and **printf** except that they take a stream pointer as their first argument. There are three predefined stream pointers for standard devices:

**stdin** For the standard input (keyboard)

**stdout** For the standard output (display screen)

**stderr** For the standard error (display screen)

RESETTING THE INPUT/OUTPUT BUFFER

All standard input and output is buffered unless otherwise stated. The **setbuf** system call with a parameter of **null** as a buffer pointer resets the standard I/O file to be unbuffered. Alternatively, **fflush** may be performed whenever a flush of output is required. The following example illustrates the use of these commands:

**setbuf(stdout, null);** Resets standard output to be unbuffered)

**fflush(stdout);** Flushes current output

# PART THREE
# ADVANCED C PROGRAMMING

# *Chapter 12*
# *Advanced System Calls*

The FOR:PRO operating system offers a number of low-level routines for the C programmer. These include interface functions for file descriptors as well as a number of operating system services. In addition to the standard **read** and **write** I/O calls, FOR:PRO offers a number of other operating system services to the C programmer. These services include commands for reconfiguring character devices (terminals), creating and overlaying processes, passing codes or sending signals between processes, manually allocating space and using larger logical pieces of disk storage (file systems).

## INTERFACE FUNCTIONS

An interface exists between C programs and the FOR:PRO operating system. A number of standard I/O function calls translate directly into operating-system traps. It is by means of such traps that C programs communicate with the FOR:PRO kernel.

The input and output routines described in Chapter 11, "Input and Output," refer to this interface as the "low level routines." Often, these routines are called directly from C code.

To gain access to a file it must first be either created or opened. In both cases, the value returned is called a file descriptor.

This file descriptor is actually an index into a process's own file table. The possible values of the file descriptor range from zero to one less than the maximum number of files open at one time during the program execution. This maximum number of files is 16.

Associated with each open file descriptor is an internal file position also called a "seek pointer." The **read**, **write**, and explicit **lseek** calls move the seek pointer throughout the file. The functions that create and use the file descriptor are:

### open: Opening a File for Reading or Writing by Function Calls

The **open** function call attempts to open a file for reading and/or writing by subsequent function calls.

```
fd = open(filename, mode);
```

The filename is a null-terminated string corresponding to the name of the file to be opened. If not preceded by an explicit directory name, the file is presumed to exist in the current directory.

Full pathnames may be used to specify the file. The **mode** indicates the mode in which the opened file is to be used. The modes are: reading (0), writing (1), and both reading and writing (2). Once opened, the file pointer is positioned at the beginning of the file. A full description of the available modes can be found in the Fortune document, Introduction to FOR:PRO.

When a process is invoked, it is given three standard file descriptors:

| File Descriptor | Opened File |
|---|---|
| 0 | Standard input |
| 1 | Standard output |
| 2 | Standard error |

These descriptors may be changed if necessary by using other function calls.

## creat: Creating a New File

Files that are needed by a program and that do not exist must be created using the **creat** function. Like the **open** function, the create function returns a file descriptor. Creating a file that already exists removes the previous contents and truncates it to zero.

Use this format for the **creat** function:

**fd = creat(**filename, mode**)**

The filename is a null-terminated string corresponding to the name of the file to be created. If not preceded by an explicit directory name, the file is presumed to exist in the current directory. Full pathnames may be used to specify the file. The mode indicates the permission bits to be associated with the file when it is created. A full description of the available modes can be found on the **chmod** and **umask** pages in the FOR:PRO Programmer's Manual.

## read: Moving Data from a File into Memory

The **read** function is used to move data from a file into program memory. A file descriptor returned from an **open** or **creat** is required.

The **read** function format is

```
bytes = read(fd, buffer, nbytes);
```

The argument fd is a file descriptor returned from an **open** or **creat** function call. Either a number of bytes equal to nbytes is transferred to the buffer address or the value -1 is returned upon failure.

## write: Moving Data from a Program to a File

The **write** function is used to move data from memory to a file. A file descriptor, returned from an **open** or **creat** call, is required.

The **write** function format is

```
bytes = write(fd, buffer, nbytes);
```

The argument fd is a file descriptor returned from an **open** or **creat** function call. Either a number of bytes equal to nbytes is transferred to the buffer address (or the value -1 is returned upon failure).

## close: Closing a File

The **close** function is used to close a file that is no longer needed for reading or writing. When an open file is closed, the file descriptor is freed for use by another **open** call. The **close** function format is

```
close(fd);
```

The fd argument is the file descriptor returned from a previous **open** or **creat** call.

## unlink: Breaking the Link Between Filename and Inode

The **unlink** function is used to break the link between a file's name and its inode in the file system. If only one link exists, the file itself is removed from the file system.

The **unlink** function format is

**unlink(**filename**);**

The filename is a null-terminated string corresponding to the name of the file to be unlinked. If not preceded by an explicit directory name, the file is presumed to exist in the current directory. Full pathnames may be used to specify the file.

## lseek: Positioning the Internal File Seek Pointer

The **lseek** function call is used to position the internal file seek pointer in a file. The function requires a file descriptor returned from an **open** or **creat** function call.

The **lseek** function format is

```
long pos;
pos = lseek(fd, offset, origin);
```

The argument fd is a file descriptor returned from an **open** or **creat** function call. The notation offset depends on the origin according to specific rules, as follows:

| Origin | Offset |
|---|---|
| 0 | Seek pointer is set to offset bytes from beginning of file. |
| 1 | Seek pointer is set to its current location plus offset bytes. |
| 2 | Seek pointer is set to size of file plus offset bytes. |

The return value is the current position in the file. It must be returned in a long variable since a file may be larger than 64KB. The value -1 is returned for an undefined file descriptor. The **seek** has no meaning when used to find a position before the beginning of a file or when used on a pipe.

These functions, like those declared in a C program, expect parameters and return values. In general, FOR:PRO system calls return the value -1 when encountering an error and 0 when there are none. The use of these functions is described on the next page. Table 12-1 lists the system calls.

Table 12-1. Principal FOR:PRO System Services

| System Call | Definition |
|---|---|
| **ioctl** | Controls a character device |
| **system** | Executes a process and wait |
| **fork** | Creates a process |
| **wait** | Waits on child process's completion |
| **exec** | Overlays a process |
| **exit** | Terminates a process |
| **kill** | Sends a signal |
| **pipe** | Sets up a pipe between two processes |
| **signal** | Sets up responses to incoming signals |
| **mount** | Mounts a logical file system |
| **umount** | Unmounts a logical file system |

ioctl: Controlling the Character Device

The **ioctl** system call performs a variety of functions on a special character file such as a terminal or a printer. The call appears as:

**ioctl(**file_des, special_code, arg_pointer**);**

The file_des is the file descriptor returned from an **open** call. The file opened should be from the device directory **/dev** and have a **c** as its first permission character (using the **ls -l** command). The **c** corresponds to a character special file.

The **special_code** is a device-specific value. The **arg_pointer** is a pointer to a list of arguments that is usually a structure.

The header file **sgtty.h** should be included when using **ioctl** because it contains the structure definition for passing arguments.

Note the following example:

```
#include <sgtty.h>

main()
{
     int fd;
     struct sgtty sgtty;

     fd = open("/dev/tty01",2);
     if(ioctl(fd, TIOCGETP, &sgtty) < 0)
          printf("ioctl returned an error\n");
     ...
     ...
}
```

**TIOCGETP** is defined as the code for getting terminal-dependent parameters and status. In this example the structure **sgtty** contains status information regarding the terminal device **/dev/tty01**, including its baud rate and special control characters such as the characters corresponding to the backspace and the line erase (kill).

system: Passing Return Codes Between Processes

The **system** call launches a new process and waits for its completion. To invoke it, use the following syntax:

```
system("process file");
```

The process file is an executable file to be run while the calling process is suspended.

Note the use of system in the following program sample:

```
smallproc:    main()
              {
                   printf("I am a small process\n");
                   exit(1);
              }

largeproc:    main()
              {
                   int retval;

                   printf("I am a large process\n");
                   retval = system("smallproc");
                   printf("Smallproc returned %d\n",
                   retval);
              }
```

**system** can be used to run any command you can type at the terminal. For example:

```
system("ls -las");
```

produces a listing of the files in your current directory on the standard output.

## fork and exec: Creating Processes

These two FOR:PRO system calls are used for creating new processes. Since they have a profound effect on the entire group of executing processes, they should be used with care.

### fork: Spawning a New Process

The **fork** system call creates a new process by making a copy of the current process core image. The resulting process is nearly identical to the original, including the files that are open. There are two differences: the processes have different process-identification numbers and they return different values from the **fork** call. The parent process fork returns the process identification of its child. The child process returns a zero.

Here is the syntax of **fork**:

```
fork();  No parameters
```

The fork call is normally used in conjunction with the **exec** call to overlay a different process on the child.

Note the use of fork in the following example:

```
...
if(fork() == 0){
     /* child code */
else{
     /* parent code */
}
```

Remember that two processes exist after the fork is executed. The example above illustrates the way to distinguish between them.

### exec: Overlaying a Process with a File to Execute

The **exec** call permits a process to overlay itself with another process started from a file. With this call, it is not possible to return to the calling process since the original core image is overwritten. Several variations of this call allow for parameters in different formats.

The following are the syntaxes of two forms of **exec**:

```
execl(name, arg0, arg1, arg2, ...);

execv(name, argv);
```

The name is the pathname of the file to be executed. In the first file call, arg0, arg1, and arg2 are pointers to strings that are the arguments to the process name. In the second call, argv is a pointer to an array of arguments.

Often, **exec** is used with **fork** to launch a completely new process from an existing one. The following example incorporates **fork**, **exec**, and **wait**.

```
int status;
...
...
if(fork() == ){

     /* child code */

     exec("newproc", "a1", "a2");

}
wait (& status);
/* more parent code */

...
...
```

The child process spawned by use of **fork** is overlaid by **"newproc"**, the new process file. The call to **wait**, if included, will cause the parent process to wait until the child finishes before more instructions are executed.

## exit: Terminating a Process

The **exit** system call terminates a process and passes back a return value. Its syntax is

```
exit(return_value);
```

The **return_value** is an integer value returned from a process that has just terminated (possibly called from a **system** request. See the use of **exit** in the example program in **wait** below.

## wait: Waiting for Descendants After an Exec

The extensively commented example program below illustrates most of the details of the **wait** call. The main use of wait (in conjunction with fork) is to wait for descendants after an exec. wait is almost always called in the parent. If there are descendant (child processes or their offspring) processes running, this call suspends the parent process until a descendant terminates. If there are no descendants, wait returns -1; otherwise, wait returns the process-id of the child which has just terminated. This process-id is useful if more than one child is executing, and the parent needs to know which child has just terminated.

wait is usually called with an integer argument, conventionally called Status. If Status is declared int, then wait is called as follows:

```
wait(&Status);
```

If Status is declared as an integer pointer, int *Status, then wait is called by:

```
wait(Status);
```

Status holds the value returned by a call to exit in the child or the number of the signal which terminated the wait.

The second exec in the program below illustrates a non-zero exit status. The program, torun, included after the main example program, calls exit with -1. The second to the lowest byte in status holds this exit Status. (See the fourth line of the output below.)

In case of an unsuccessful exec, the value returned (in the integer variable status) is undefined unless the child process explicitly calls exit. This is illustrated in the third exec below where the fictitious routine los is "exec"ed.

If wait is terminated by a signal instead of by the termination of a child, the lowest byte in Status contains the signal number. You can see this by running the program and hitting the Cancel key at different times.

The example program below produces the following output:

```
12 -rwxrwxr-x 1 fisher  11989 Jan 18 16:41 waitTest
waited for child PID 2703, exit value 0, 'signal' 0x0
Why can't I be as wise as Boethius?
waited for child PID 2704, exit value -1, 'signal' 0x0
fictitious  PID is 2705
execlp failed
waited for child PID 2705, exit value 1, 'signal' 0x0
```

```
#include <signal.h>
#include <stdio.h>

  /* this function is used to catch signals */
catch_int()
{
    printf("got interrupt in parent\n");
}

main()
{
      /* pid_child holds the process-id of the child; Status holds
         the value returned by exit(). */
    int pid_child, Status;
    char status2;

      /* catch signals IF they are not already being ignored. We
         do this because some shells (like Bourne) set background
         processes to ignore keyboard interrupts. */
    if(signal(SIGINT,SIG_IGN) != SIG_IGN)
        signal(SIGINT, catch_int);

      /* first exec, the program splits into two "identical pieces" */
    pid_child = fork();
    if (pid_child  == 0) {
          /* the exec is done only in the child */
        printf("ls PID is %d\n",getpid());
        execlp("ls","ls","-las", "waitTest",0);
        printf("execlp failed \n");     exit(1);
    }

    else {
          /* the process-id is not zero in the parent */
          /* wait returns the process-id of the terminated child */
        pid_child = wait(&Status);
          /* the tricky part here is that the status is returned
             in the second lowest byte not in the highest byte as
             it says in many man pages. The highest byte syndrome
             is left over from pdp-11 days and is a source of many
             bugs. We right shift Status by 8 bits and then AND
             the result with 0xff which strips off the significant
             byte which is put into a char,status2; In addition,
             wait can terminate because a signal has been received,
             for example, from the keyboard.  You can see this by
             interrupting this program by hitting cancel/del. When
             this happens, the low byte of status contains the
             signal number. */
        status2 = (Status>>8)&0xff;
        printf(
          "waited for child PID %d, exit value %d, 'signal' 0x%x\n",
           pid_child, status2,Status&0xff);
        fflush(stdout);
    }
```

```
     /* second exec */
    pid_child = fork();
    if (pid_child  == 0) {
        printf("torun  PID is %d\n",getpid());
        execlp("torun","torun",0);
        printf("execlp failed \n");      exit(1);
    }
    else {
        pid_child = wait(&Status);
          /* torun returns non-zero exit status */
        status2 = (Status>>8)&0xff;
        printf(
          "waited for child PID %d, exit value %d, 'signal' 0x%x\n",
           pid_child, status2,Status&0xff);
        fflush(stdout);
    }
      /* third exec with non-existent process */
    pid_child = fork();
    if (pid_child  == 0) {
        printf("fictitious  PID is %d\n",getpid());
        execlp("los","los","-las",0);
          /* we can only get here if the exec fails */
        printf("execlp failed \n");     exit(1);
        fflush(stdout);
    }
    else {
        pid_child = wait(&Status);
        status2 = (Status>>8)&0xff;
        printf(
          "waited for child PID %d, exit value %d, 'signal' 0x%x\n",
           pid_child, status2,Status&0xff);
    }
}
```

The next little program is torun which is exec'ed in the second exec above.

```
main()
{
    printf("why can't I be as wise as Boethius?\n");
    exit(-1);
}
```

## signal and kill: Using Process Signals

FOR:PRO processes may send and receive software interrupts or signals. These signals do not carry any infcrmation other than their signal number. They are used to indicate particular process conditions. The file **/usr/include/signal.h** contains the definitions for the signals and should be included when using **signal** and **kill**.

## signal: Preparing to Receive a Signal

This system call is used to indicate the routine to be executed when a catchable signal is detected. Upon receiving the signal, the process is suspended and execution continues with the indicated signal routine. When the routine exits, the process resumes from the point of interruption.

The following is the syntax of **signal:**

```
signal(sig_number, routine);
```

The **sig_number** is the number of the signal being caught and the **routine** is the function to be executed.

Note the use of signal in the following example:

```
#include <signal.h>

signal(SIGINT, intfunc);
...
...
...

intfunc()
{
     signal(SIGINT, intfunc);
     printf("Received a keyed interrupt\n");
}
```

Pressing CANCEL generates the SIGINT interrupt signal. In this example, the program catches the signal and prints a message. Notice that the signal-catching routine must call **signal** again to catch the next signal. This signal-catching routine is reset each time a signal is caught. See the example programs in the material on the pipe system call and the wait system call.

## kill: Sending a Signal

The name of this call is misleading; **kill** merely signals to processes. Its syntax is

```
kill(pid, sig_number);
```

The pid is the process-id of the process to receive the signal. The sig_number is the signal number to be sent, as defined in **/usr/include/signal.h.** Note that the sending and receiving processes must be owned by the same user or the signal is not sent.

## pipe: An I/O Channel Between Two Processes

This section introduces the pipe system call by means of an extensively commented example. In addition to illustrating the pipe system call, the example illustrates the following system calls:

- fork
- exec
- exit
- wait
- signal

```
ls -las | more
```

The command above "pipes" the output of the **ls** command to the input of the **more** command. Instead of sending its output to the standard output, **ls** sends its output to the pipe. The command **more** reads its input from the other end of the pipe instead of from a file given as an argument. The shell uses the pipe system call to allow **ls** and **more** to communicate in this way.

The pipe system call takes a two-element integer array as an argument and returns two file descriptors. The first element holds the file descriptor for the **read** side of the pipe; the second element holds the file descriptor for the **write** side. These file descriptors allow user processes to **write** to and read from an 8,192 element buffer.

Writing is suspended when this buffer is full.

When the write end of a pipe is closed, any process that reads it, will read an end of file.

The two ends of a pipe are named according to the way they are used. A process writes characters into the write end of a pipe and reads characters from the read end.

The example below uses the pipe system call to set up a communication channel between two child processes. The output of the **ls** process is piped into the input of the child **more** process.

```
/* comments generally describe the statement immediately below them */
#include <signal.h>
#include <stdio.h>

#define READ 0
#define WRITE 1

static int pipeopen_pid;

/*  the following procedure takes a command cmd and a
    mode which is either READ or WRITE.  It next calls the
    pipe command and then creates a child process which exec's
    the command cmd.  pipeopen returns a file descriptor
    which can be used to read or write the pipe.  */

pipeopen(cmd. mode)
char *cmd;
int mode;
{
        /*   the pipe command takes an array of two file descriptors
        p[0] holds the read side file descriptor and p[1] holds
        the write side file descriptor. */
    int p[2];

        /*   the call to the pipe command */
    if (pipe(p) < 0)
        return(NULL);
        /*   in the next statement we split into two processes.  */
        /*   child code follows  */
    if ( (pipeopen_pid = fork()) == 0) {
        /*   close the write side of the pipe in the child */
    close(p[WRITE]);
        /*   the next statement closes the standard input
             of the child and puts the pipe's read file
             descriptor in its place.  This allows more
             , which reads from the standard input, to get
             its input from the read side of the pipe.  */
    dup2(p[READ],0);
        /*   the execl executes the command */
    execl("/bin/sh", "sh", "-c", cmd, 0);
    _exit(1); /* disaster has occurred if we get here */
    }
        /*   end of child code  */

    if (pipeopen_pid == -1)
        return(NULL);
        /*   since another child process is going to write into
        the pipe, we close the parent's read side  */
    close(p[READ] );
        /*   Next we return the pipe's write side file descriptor.  */
    return(p[WRITE]);
}
```

```
pclose(fd)  /*    close the pipe fd */
int fd;
{
    register r, (*hstat)(), (*istat)(), (*qstat)();
    int status;

    close(fd);
       /*     save the signal values */
    istat = signal(SIGINT, SIG_IGN);
    qstat = signal(SIGQUIT, SIG_IGN);
    hstat = signal(SIGHUP, SIG_IGN);
       /*     wait until the child processes terminates */
    while ((r = wait(&status)) != pipeopen_pid && r != -1);
    if (r == -1)
         status = -1;
       /*     the next three statements sets the signals back
         to their previous values.  */
    signal(SIGINT, istat);
    signal(SIGQUIT, astat);
    signal(SIGHUP, hstat);
    return(stgatus);
}

main ()
{
    int status;
         /*   fout will hold the pipe's write fd */
    int fout;

         /*   get the file descriptor for the write side of the pipe */
    fout = pipeopen("more", WRITE );
         /*   next fork off a child process to do the ls "  */
    if (fork() == 0)  {
         /*   dup2 first closes the standard output file
              descriptor and then moves the pipe's write
              file descriptor to the standard output. */
         dup2(fout, 1);
          /*  If you remove the next statement, the program
              will not work.  It will hang.  To get the
              output, you will have to type control-backslash
              This statement closes fout so that pclose below
              will flush the pipe */
         close(fout);
         /*   The next statement exec's the ls command  */
         execlp("ls","ls", "-las", 0);
         printf("execlp of ls -las failed \n");
         exit(0);
    }
```

```
    /*  pclose flushes out any remaining characters in the
        pipe.  When more reads the closed pipe, it will
        get an end of file.  Note, more reads the keyboard
        by reading /dev/tty.  */

    pclose(fout);
}
```

mount and umount: Mounting and Unmounting File Systems

Entire physical disk drives may be divided into one or more logical file systems. Normally, the logical files do not overlap.

One of these file systems, the **root /**, is used by the kernel. It is always mounted and is home for the system commands and utilities. Other file systems (if any exist) may be mounted and unmounted practically anywhere in the root file system tree as follows.

mount: Mounting a File System on a Directory

This call notifies FOR:PRO that a file system is being mounted in the root file system tree. It is invoked by:

**mount(**special_file, name, rwflag**);**

The special_file is a block-structured device file found in the **/dev** directory. The name is the pathname of the directory on which the mountable file system is to be placed.

Note that the previous contents of this directory become inaccessible while the file system is mounted. They will reappear when the file system is unmounted. The rwflag indicates whether the file system is to be mounted for reading only.

umount: Unmounting a Mounted File System

This call unmounts a previously mounted file system. Its syntax is:

**umount(**special_file**);**

The only parameter for this call is the block-structured device file from the **/dev** directory used in a previous mount command. This call will fail if a user is currently in a file or directory or accessing a file in the mounted file system.

# *Chapter 13*
# *Function Calling Conventions*

In advanced programming applications, it is often necessary to understand how and where symbolic names and their data are stored in physical memory. These are governed by the conventions for the placement of frames on the run-time stack and for data alignment by the compiler. These coventions are described below.

## THE C RUN-TIME STACK

Whenever a C function is called, an activation record or frame is pushed onto the run-time stack. This frame is used to pass arguments to and from the function, to save the calling function's registers and return address, and to provide space for the called function's local variables.

The frame pointer (fp) identifies the current activation record for the function being executed. (See Figure 13-1.) The stack grows downward. Arguments are pushed in reverse order, so argument 1 is pushed last and argument N is pushed first. The first argument is located at fp+8 on the stack.

When calling and returning from C functions, the calling function and the called function must perform certain operations in conjunction with the run-time stack. These operations are described as follows.

### Entry Sequence

On entry to a function, the calling function's arguments and return address will have been pushed onto the run-time stack frame.

Note that the number of arguments pushed must correspond exactly to the number of arguments expected. If the number of arguments in each call doesn't correspond, the stack will be unbalanced and will probably produce an invalid result.

The called function will immediately save the old frame pointer register on the stack and then update the register to point to the new stack. Local variable and register save space is allocated, and the registers are saved in that space. Then execution of the called function begins.

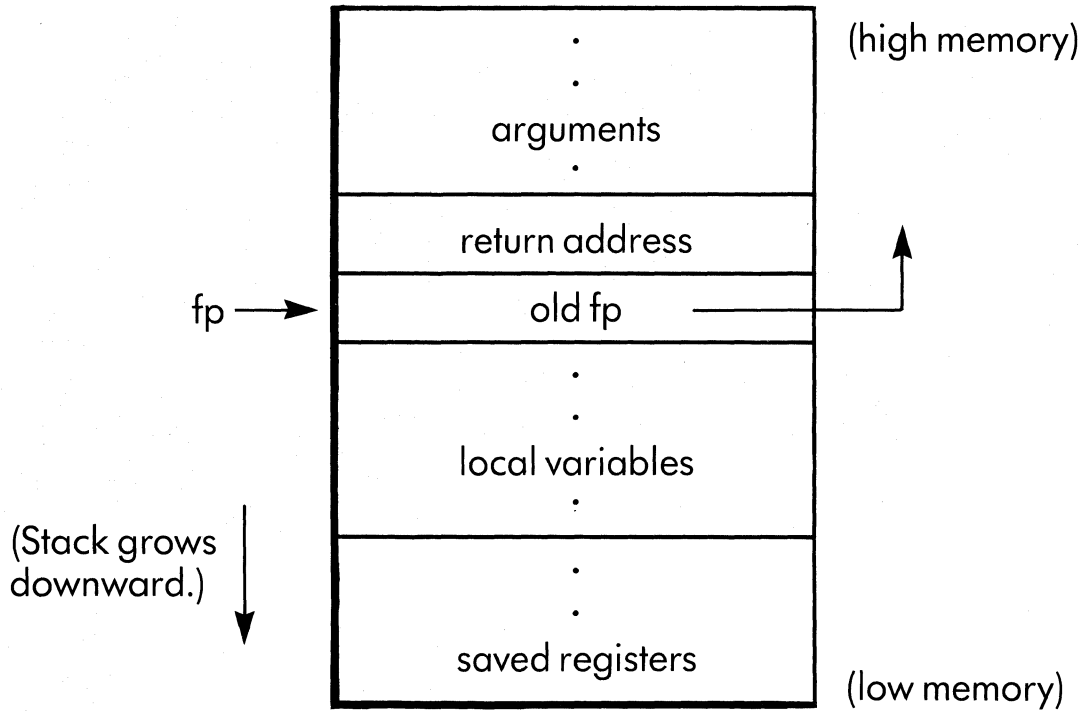


Figure 13-1. The Run-Time Stack (Stack Grows Down the Page, or From High Memory to Low Memory)

### Exit Sequence

Upon exit from a function, the reverse operations of entry are performed. The called function restores all registers including the old frame pointer and old stack pointer. The activation record or stack frame is removed. Control is passed back to the return address of the calling function. The calling program unit clears the arguments from the stack it originally pushed.

## DATA ALIGNMENT

All variables are aligned by the **cc** compiler on even byte addresses in memory. If a data structure contains an odd number of bytes, it is padded to the next even address with an extra zero byte.

If you are attempting to maintain machine independence for portability, keep in mind that the lengths of the different data types vary. Since future releases of the Fortune C compiler may define the sizes of the various data types differently, the different data types and the amount of storage allocated each should be respected. The size of operators in C allows you to avoid using absolute numbers when referring to particular data objects.

Addressing should not be performed within a basic type. It is dangerous to set a pointer of one size to the address of a variable of another size to access different portions of the basic type. The **union** construction should be used instead.

The following example shows an incorrect method of setting pointers:

```
long longvar;
short s1,s2;
short *shortp;

shortp = (short *)&longvar;
s1 = *shortp;
shortp++;
s2 = *shortp;
```

The above program fragment should be replaced by:

```
union {
     long longvar;
     struct {
          short s1;
          short s2;
     }shortvars;
}l_to_sh;
```

In this example, the variable **l_to_sh.longvar** is overlaid by the two short variables **l_to_sh.shortvars.s1** and **l_to_sh.shortvars.s2.**

Addresses should not be calculated within a structure. The fields of a structure should be used explicitly to access data within that structure.

Because variables are aligned on even boundaries, extra zero bytes may pad structures. Incrementing a pointer through a structure may fail to reveal padding bytes inserted by the compiler.

Local variables should be accessed only with their declared names. Because local variables are allocated on the run-time stack, a pointer moving through these locals can inadvertently modify other stack information required for linking the procedure to the calling program. The result may produce errors that are extremely difficult to find.

The Motorola 68000 processor addresses bytes sequentially from high to low order. If a pointer to an integer is cast to a pointer to a character, the first character referenced will be the high order byte of the integer (that is, the most significant byte).

An example is the following program:

```
main()
{
        char *cp;
        short i = 259;        /* i = 0x0103 */

        cp = (char *)&i;

        printf("1st char = %d\n",*cp);
        cp++;
        printf("2nd char = %d\n",*cp);
}
```

which produces the output:

```
1st char = 1
2nd char = 3
```

# *Chapter 14*
# *Linking Programs in Different Languages*

With the FOR:PRO operating system, it is possible to call routines or functions written in one language from programs written in another language. For example, you can write a program in C that calls a math package in FORTRAN 77 or a graphics package in Pascal.

To call functions written in different languages, you need to compile and link the functions and your C program separately as described in Chapter 2. You also need to observe special conventions for declaring data and functions, for including libraries, and for actually calling a procedure. These conventions are summarized in this chapter.

## DECLARING DATA

Table 14-1 compares the sizes of variables declared in the C, FORTRAN 77, and Pascal languages. As this table indicates, data types of some languages do not exist in others. The unit of size used for comparison is the byte.

Table 14-1 allows you to determine the proper data declaration for passing parameters when one language issues a call to another.

## DECLARING FUNCTIONS AND PROCEDURES

Table 14-2 shows examples of equivalent function declarations (which are called "definitions" in C) in the three Fortune System languages. Table 14-3 provides examples of equivalent procedure declarations.

## INCLUDING ADDITIONAL LIBRARIES

When you call subroutines written in a language different from your programming language, you may need to include additional libraries when linking your program. For example, if you are compiling a C program that calls a Pascal procedure, you need to use the **-lpc** and **-lm** option when invoking cc (C compiler). If you are compiling a Pascal program in which a FORTRAN subroutine is called, you need to add the **-lI77** and **-lf&7** options as well as the **-lm** option.

Table 14-1. Physical Data Representations

| Size in Bytes | FORTRAN 77 | Pascal | C |
|---|---|---|---|
| 2 | integer*2 | NONE | short |
| 4 | integer | integer | long,int |
| 4 | real | NONE | float |
| 1 | character | char,boolean | char |
| n | character*n | char | char |
| 4 | logical | NONE | NONE |
| 8 | complex | NONE | struct{float r,i} |
| 16 | double complex | record | struct{double dr,di} |
| 8 | double precision | real | double |
| 2 or 4 | statement label | NONE | NONE |
| varies | hollerith | NONE | NONE |

Table 14-2. Equivalent Function Declarations

Pascal
```
type realptr = ^real;
function power(var x:real;var y:integer):real;
forward (or external);
```

C
```
double power(x,y)
double #x;
int #y;
```

FORTRAN
```
double precision function power(x,y)
double precision x
integer y
```

Table 14-3. Equivalent Procedure Declarations

| | |
|---|---|
| Pascal | **procedure minmax(var x,y:integer);** |
| FORTRAN | **subroutine minmax(x,y)**<br>integer x,y |
| C | The concept of a procedure does not exist as such in C; the C function is equivalent to the FORTRAN function or subroutine and the Pascal procedure.) |

## INCLUDING ADDITIONAL LIBRARIES

When you call subroutines written in a language different from your programming language, you may need to include additional libraries when linking your program. For example, if you are compiling a C program that calls a Pascal procedure, you need to use the **-lpc** and **-lm** option when invoking cc (C compiler). If you are compiling a Pascal program in which a FORTRAN subroutine is called, you need to add the **-lI77** and **-lf&7** options as well as the **-lm** option.

In the case of I/O modules, it is always advisable to use the I/O of the language you are compiling. In other words, if you are compiling with cc (C compiler) and linking a routine written in FORTRAN 77, the I/O library of C should be used.

## CALLING A PROCEDURE

Calling conventions vary from one language to another. The following example illustrates calling a FORTRAN 77 function from a C program.

C Program:

```
/*
 *  This C program calls a FORTRAN 77 function
     *  to raise a number to a given power.
 *  Since FORTRAN 77 passes its arguments
 *  by reference,  an & must precede each variable.
 */

main()
{
     int i;
     int y = 3;

     for(i = 1; i <= 10; i++)
```

```
        /* calculate i cubed */

             printf("%d raised to %d equals %d\n",
                  i, y, power(&i, &y));
    }
```

FORTRAN 77 Function:

```
    integer function power(base, exponent)
    integer base, exponent

    power = base ** exponent
    return
    end
```

# *Appendix A*
# *Compiler Passes, Options and Functions*

This Appendix provides detailed information on the individual compiler passes and the options that can be used with the compiler that are introduced in Chapter 1. A description of compile-time functions for reading files and for defining global constants and macros is also included.

## THE COMPILER PASSES

The six compiler passes are:

### The Preprocessor

The C preprocessor interprets commands preceded by the **#** character when encountered in the source code. These commands permit files to be included (**#include**), macros to be defined and substituted (**#define**), and different sections of the code to be output. The output of the preprocessor alone may be generated and placed in **std.out** by using the **-E** compiler option.

### The C Compiler

The C compiler parses the C source code and generates assembly language output. The assembly language output of the compiler may be preserved by using the **-S** option. It is saved in a file suffixed by **.s**.

### The C Optimizer

A peephole (as opposed to global) optimizer is available as an option to the C compiler. It may be invoked by using the **-O** command option to **cc**. It will reduce code 20-25 percent in size, allowing it to run faster. For example:

```
$ cc -O file.c
```

Produces an optimized code.

## The Fast Floating-Point Processor

The fast floating-point processor /usr/lib/ffp takes the assembly language file generated by the compiler and converts all floating point operations and constants to a much faster single-precision floating-point form. The resulting file is slightly larger and suffers a loss of precision in return for much faster floating-point operations.

## The Assembler

The FOR:PRO assembler **ac** is automatically invoked as part of the C compiler. It accepts input files suffixed with **.s** and generates relocatable object modules suffixed with **.o**. For example:

```
$ /usr/lib/ac test.s
```

produces the object module **test.o**.

Remember that using the **cc** command on an assembly source file will automatically call the assembler followed by the linking loader.

## The Linking Loader

The linking loader **ld** binds one or more relocatable objects together with optional libraries to create an executable object module. Provided the **-c** option is not used to generate an intermediate relocatable object file, the C compiler automatically calls the loader. If the **-c** option is used, the loader may be invoked as follows:

```
$ ld /usr/lib/crt0.o test.o /usr/lib/libc.a -o test
```

## OPTIONS USED WITH cc COMMAND

The C compiler options are introduced in Chapter 1 of this guide. The following examples provide additional instruction in their use.

| Option | Purpose |
|---|---|
| **-c** | Creates a relocatable object file **.o** using the same name as the source file. For example<br>`$ cc -c file.c` |

| Option | Purpose |
|---|---|
| | produces:<br><br>**file.o**<br><br>This example produces a relocatable object file named **file.o** instead of the executable file **a.out**. Since the loader has not been invoked, this **.o** file is not executable. The **.o** file may be used on the command line of a subsequent compilation. |
| **-o** | Names the output file. For example<br><br>**$ cc -o** test test.c<br><br>produces<br><br>test<br><br>This command enables you to give your own name to the executable object file. Except for the name, test is exactly the same as **a.out**. |
| **-O** | Calls the peephole optimizer. For example:<br><br>**$ cc -O** file.c<br><br>The **a.out** file produced will be smaller in size and run faster. |
| **-v** | Lists the compiler passes as they occur. For example:<br><br>**$ cc -v** test.c<br><br>produces the output<br><br>/usr/lib/cpp -Dmc68000 -Uvax test.c /tmp/ctm001044.<br>/usr/lib/ccom < /tmp/ctm001044.s > /tmp/ctm001043.s<br>/usr/lib/ac -o test.o /tmp/ctm001043.s<br>/usr/bin/ld /usr/lib/crt0.o test.o /usr/lib/libc.a |
| **-G** | Turns off stack checking. For example:<br><br>**$ cc -G** test.c<br><br>This option decreases the next (code) size. If the stack is not used extensively, this decrease produces a slight improvement in running time. |

| Option | Purpose |
|---|---|

**CAUTION:** Do not use the **-G** option with programs that allocate more than 8KB of stack space or with programs that use recursion.

Example:

```
/*
*  Since local variables are allocated in the stack,
*  the following example requires 4000 * 4 bytes of
*  stack space.
*/

main()
{
     int x[4000],i;

     for (i = 0;i < 4000;i++)
          x = 0;
}
```

**-E** Runs only the preprocessor. For example,

```
$ cc -E test.c
```

produces the output:

```
#1 test.c (text of the program)
```

**-C** Prevents the preprocessor from removing comments. For example:

```
$ cc  -E  test.c     (Comments removed.)

$ cc  -E  -C  test.c      (Comments remain.)
```

**-Dname** Defines an identifier name the preprocessor.

**-Dname=def** Defines an identifier name and assigns it the value **def**. For example:

```
$ cc  test.c  -DSIZE
```

When a value is omitted, a default value of 1 is used. The constant SIZE above is defined and assigned the value 1.

| Option | Purpose |
|---|---|
| | Another example:<br><br>`$ cc test.c -DSIZE=12`<br><br>The constant **SIZE** is defined and assigned the value 12. |
| **-S** | Generates assembly language output.<br><br>The assembly language output of the compiler is preserved in a file suffixed by **.s**. For example:<br><br>`$ cc -S test.c`<br><br>produces<br><br>test.s |
| **-Uname** | Eliminates an identifier name that has been defined for the preprocessor. For example:<br><br>`$ cc test.c -USIZE` The constant definition SIZE is eliminated. |
| **-Idir** | Searches for **#include** files in the **dir** directory. For example:<br><br>`$ cc -I/usr/newlib test.c`<br><br>A search is made through several standard directories for any **#include** files listed in the C source file **test.c** that are not full pathnames (do not begin with **/**). Typically, the current directory is searched first. If the files are not found here, the system directory **/usr/include** is searched. The **-I** option in the example above causes the **/usr/newlib** directory to be searched first. If the files are not found there, the other directories are searched in the usual sequence. |
| **-g** | Produces additional symbol-table information for the Fortune symbolic debugger. |
| **-w** | Suppresses warning messages. |
| **-p** | Prepares the executable program for use with the profiler. |

## COMPILE-TIME FUNCTIONS

The C language preprocessor permits a C program to read files at compile-time, to define global constants and macros, and to compile different source code based on global definitions. These compile-time functions are used as follows:

### #include

The **#include** command reads and expands a UNIX file inline during compilation.

The **#include** command format is

```
#include "filename"
```

When double quotes " are used, the current directory is searched for **filename** before the standard directories. When the command and greater than , > signs are used, only the standard directories are searched.

### #define

The **#define** command associates a name with a constant. These are examples of the **#define** command:

```
#define N  100
#define M   N-1
```

The **#define** command may also define a macro substitution in this way.

```
#define MASK(i)  i &= 0377
```

In the above example, semicolons have intentionally been omitted. As in a C function call, semicolons are automatically provided when this macro is used.

```
MASK(a);   becomes   a &= 0377;
```

### #if and #ifdef

Sections of a C program may be conditionally compiled using the **#if**, **#ifdef**, and **#ifndef** commands if those sections of code are appropriately surrounded.

```
#if 32 16          The enclosed C program section
...                compiles if constant
...                32:16 is non-zero or true.
...
#endif
```

```
#ifdef 32 16        The enclosed C program section
...                 compiles if constant
...                 32:16 appears in a previous
...                 define command or in a
#endif              -D runtime option of the compiler.

#ifndef  32 16      The enclosed C program compiles
...                 if constant 32:16 is NOT defined
    ...             or is a -U runtime option of the
    #endif          compiler.
```

The command **I#else** may be used similarly to the **else** for the alternative code in a conditional compilation.

# *Appendix B*
# *List of C Language Files*

The following files are included on the C product disk.

In directory: /f/usr/bin:

| | |
|---|---|
| **cb** | C beautifier |
| **cc** | C compiler (driver) |
| **lex** | Lexical analyzer generator |
| **lint** | C semantic checker (types,...) |
| **yacc** | Yet another compiler compiler (parser generator) |

In directory: /f/usr/lib:

| | |
|---|---|
| **ccom** | C compiler |
| **libl.a** | link library |
| **libln.a** | lint library |
| **lint1** | pass one of lint |
| **lint2** | pass one of lint |
| **llib-lc** | lint library |
| **llib-lc.ln** | lint library |
| **llib-port** | lint library |
| **llib-port.ln** | lint library |
| **ncform** | part of lext |
| **yaccpar** | part of yacc |

# Appendix C
# Generating Assembly Language Output

The following programs illustrate the use of the -S compiler option to generate assembly language output.

```
/*      These programs demonstrate the calling conventions for making *
*       a subroutine call from a C program.                           */

int base = 10;                          /*  global variable "base" initialized
                                            to 10       */

main()                                  /*  main procedure      */
{
        int     result, subr();         /*  declare result, subr()      */
        int     x = 25;                 /*  declare int x, initialize       */

        result = subr(x, base);         /*  assign value returned by        */
}                                       /*  by subr() to result       */

int     subr(p1,p2)                     /*  subroutine - returns an int, 2
                                        /*  parameters      */
int     p1,p2;                          /*  parameter declarations      */

        int   temp;                     /*  declare automatic variable
                                            "temp"      */

        temp = p1 * 100 / ps;           /*  perform calculation     */
        return(temp);                   /*  return result of operations       */

LLO:
        .data
        .even
        .globl base

base:                                   |*  storage assigned for
|       line 6, file "call.c"           |*  global variable "base"
        .long  10                       |*  initialized to 10
        .text
        .globl main
main:                                   |*  start of main procedure
|.proc
        jsr   _csavl:                   |*  jump to stack-checking routine
        link  %a6,#0                    |*  save old frame pointer
        add1  #-.F1,%sp                 |*  allocate storage area for
                                        |*  local variables (automatics)
        movem1  #,S1,%sp@               |*  save address and data registers
        movb  #.FR1,$d1                 |*  load floating point reg. mask
        jsr   _regsav                   |*  save floating point registers
```

```
|A1 = 8
|       line 11, file "call.c"
        movl  #25,$a6@(-8)          |*  initialize automatic "x" to 25
|       line 13, file "call.c"      |*
        movl  base,%sp@-            |*  push arguments (last arg first)
        movl  %a6@(-B),%sp@-        |*  onto stack
        jsr  subr                   |*  jump to subroutine
        addql  #8,%sp               |*  adjust stack (clear arguments)
        movl  %d0,%a6@(-4)          |*  assign return value (returned in
        bra  .L13                   |*  $d0) to automatic "result"
L13:
        moveml  %a6@(-.F1),#,S1     |*  restore old register contents
        unlk  %a6                   |*  restore old frame pointer
        rts                         |*  end

F1 = 8                              |*  size of local storage area on stack
S1 = 0                              |*  register save mask
FR1 = 0                             |*  floating point register save mask
|m1 = 8
        .data
        .text
        .globl subr
subr:                               |*  start of subroutine "subr"
|.proc
        jsr  _csavl                 |*  stack-checking
        link  %a6,#0                |*  save frame pointer (from main)
        addl  #-.F2,%sp             |*  allocate local storage area -
                                    |*  parameters are stored here
        moveml  #,S2,%sp@           |*  save registers
        movb  #.FR2,%d1
        jsr  _regsav                |*  save floating point regs
|A2 = 16
|       line 22, file "call.c"     |*  push THREE arguments onto stack:
        movl  %a6@(12),%sp@-        |*  first "p1" for division later
        movl #100,%sp@-             |*  constant (100) and "p2" are
        movl  %a6@(8),%sp@-         |*  pushed for multiplication
        jsr  1mul                   |*  jump to multiplication routine
        addql  #8,%sp               |*  adjust stack for TWO arguments -
                                    |*  still is ONE arg (p1) on stack
        movl  %d0,%sp@-             |*  push mult, result (in %d0)
        jsr  1div                   |*  onto stack and jump to divide
        addql  #8,%sp               |*  adjust stack, clearing all args
        movl  %d0,%a6@(-4)          |*  store result in automatic "temp"
        line 23,  file "call.c"     |*
        movl  %a6@(-4),%d0          |*  move "temp" into %d0 for return
        bra  .L15
L15:
        movml  %a6@(-.F2), #.S2     |*  restore registers (not $d0)
        unlk  %a6                   |*  restore old frame pointer
        rts                         |*  return to calling routine
```

```
Fs = 4                          !*  size of local storage area on stack
S2 = 0                          !*  register save mask
FR2 = 0                         !*  floating point register save mask
! Ms = 12
        .data
```

# PLEASE GIVE US YOUR RESPONSE TO THIS MANUAL

You can help us provide manuals that suit your needs by filling out and returning this form. When a new edition of this manual is prepared, we will try to use your suggestions.

Write the name of the manual you are commenting about here _____

_____

1. Does this manual give you the information you need? Yes No
   Is any information missing?

2. Is this manual accurate? Yes No
   Please list the inaccurate information.

3. Is the manual written clearly? Yes No
   What areas are unclear?

4. What other comments about this manual do you have?

5. What do you like about this manual?

On a scale of 1 to 10, how would you rate this manual? Please circle one.
Excellent 10 9 8 7 6 5 4 3 2 1 Poor

**Name** _____ **Phone number** _____
**Company** _____
**Address** _____
**City** _____ **State** _____ **Zip Code** _____

Fortune Systems Corporation has the right to use or distribute this information as appropriate with no obligation.

1002348-02

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**
First Class Permit No. 29 San Carlos, CA

**FORTUNE SYSTEMS CORPORATION**

Attn: Publications Department
101 Twin Dolphin Drive
Redwood City, CA 94065

# USER RESPONSE CARD

## FOR FORTUNE SYSTEMS MANUALS