New C*
Reference Manual

INTERACTIVE

product family

INTERACTIVE

A Kodak Company

LPI-C*
User's Guide

This guide provides specific information for using LPI-C Version 1 on an Intel*80386*System under UNIX*/XENIX*/AIX.*

Contents

Chapter 2: LPI-C Language Specifics

Appendices

Index

Preface: Using This Guide

Product Description

LPI-C is a fully conforming implementation of the C language as defined by the ANSI Committee X3J11 in *Draft Proposed American National Standard for Information Systems--Programming Language C, (Document No X3.159-1989, Dec 7, 1988).* It passes the Plum Hall ANSI C Validation Test Suite and incorporates preprocessor improvements and an array of new features.

The *LPI-C User's Guide* describes how to compile, link, and run LPI-C programs on your system. LPI extensions to the C language are also explained in this guide.

Related Documentation

For additional information on LPI-C, refer to the following manuals in the LPI-C documentation set:

- The *LPI-C Language Reference Manual*, which describes the LPI-C language.

For information on using CodeWatch, refer to the *CodeWatch Reference Manual.*

LPI-C Features

The following LPI-C features enhance performance and productivity:

- Standard libraries and header files are packaged with LPI-C.

- Function prototypes are incorporated. These form the basis for descriptive diagnostics concerning the number and types of arguments in a function call.
- Compile-time string concatenation is provided as a convenient means of using long string literals.
- New preprocessing directives and operatives are available, including stringize, token-paste, `#elif`, `#error`, and `defined ()`.
- Execution is improved by in-line code generation of many standard library, math string/memory, and copy/compare functions.
- The `const` type qualifier designates data objects as non-modifiable, further enhancing reliability and maintainability of code. LPI-C will reliably generate diagnostics if attempts are made to modify such objects.
- The `volatile` type qualifier designates references to variables that should not be "optimized-out." This is useful, for example, when writing a hardware driver or multitasking applications.
- Automatic initialization of aggregate types (structures, unions, and arrays) within blocks is an additional feature of LPI-C, as is union initialization.
- A standard generic pointer type, `void *`, is supported.
- Cross-language calling allows you to incorporate subprograms written in other LPI languages into your LPI-C program, and vice versa. This means that you target the best language to the programming task. Also, subroutines do not have to be rewritten, thus saving developers' time for new programming tasks.
- Informative error messages flag errors by placing a pointer at the exact location of the programming error within a line.
- A full set of listing options provides annotated listings of the source program, user symbols and their attributes, a cross-reference, and a summary of compilation statistics. These all improve development efficiency.
- LPI-C is fully supported by CodeWatch, the interactive source-level debugger, which allows testing and debugging in the C language, without dependence on the machine language of the computer.

Intended Audience

This guide is intended for experienced C programmers, rather than for beginners. A working knowledge of the UNIX operating system is assumed.

Organization of Information

Chapter 1, "Using LPI-C," explains how to use LPI-C on your system.

Chapter 2, "LPI-C Language Specifics," describes the LPI-C language implementation for your system.

Appendix A provides information on installing, listing, and removing LPI software.

Appendix B describes the LPI-C compiler error messages.

Appendix C provides information on implementation-defined behavior.

Appendix D discusses compatibility issues.

The index provides a quick reference for finding important LPI-C terms.

Syntax Conventions

The following syntax conventions are used in this guide:

1. Brackets enclose optional command line entries. For example,

   lpild [*option*] ... *object_file* [*object_file*] ...

2. Variable information is printed in italics. For example,

   lpiadmin install [*option*] ... *product_name*

3. Typewriter font is used for keywords, reserved words, and where identifiers from programming examples are referred to in the text.

```
#include <stdio.h>
main ()
{
printf("This is typewriter font")
}
```

4. Optional repetition is indicated with ellipses (...). For example,

lpicc [*option*] ... *filename* [*option*]

Chapter 1: Using LPI-C

Chapter 1: Using LPI-C

Installing the LPI-C Compiler

To install LPI-C, you must first transfer the contents of the release media to a temporary directory. Your product Release Notes list the necessary steps. When the release media files reside in a temporary directory, you can begin the installation procedure, moving the files to their permanent location on your system. See Appendix A for complete details.

Note
An online copy of the Release Notes for this product can be found after installation in the directory /usr/lib/LPI in the file c__zzzzzz.info, where zzzzzz is the version number of the release.

The lpiadmin Utility

The lpiadmin utility installs LPI-C and the necessary files (components) in the appropriate directories on your system.

The lpiadmin utility keeps track of all versions of LPI products and components currently on your system and allows you to easily install, update, list, and remove LPI products. By default, the permanent location of lpiadmin is in the LPI administrative directory, /usr/lib/LPI, where it is referenced when invoking lpiadmin check, lpiadmin help, lpiadmin list, and lpiadmin remove after installation is complete. Refer to Appendix A for further information on these commands and available options.

If you have more than one LPI product on your system, different versions of the same product component (file) may exist. The runtime libraries and the compiler itself are examples of product components. The lpiadmin utility organizes LPI products on your system, allowing you to access either the latest versions of LPI components on your system or specific versions delivered with a particular product. Refer to the "Linking a Program" section in this chapter for further information.

Entering a Source Program

To enter an LPI-C source program, use any standard text editor, such as vi. For an explanation of vi commands, refer to the description of vi in your UNIX documentation.

Compiling a Program

LPI-C consists of two executable programs: the compiler, named newc, which includes an integrated macro preprocessor, and a command line processor, named lpicc.

newc allows only one filename, only LPI compiler options, and requires a separate lpild/ldc step. lpicc allows multiple filenames and accepts both cc and lpcc compiler options. The purpose of lpicc is to provide compatibility with existing make files, although lpicc may be used directly from the command line.

lpicc processes most cc options in one of the following three ways:

- lpicc executes the option in a way similar to the way cc executes it
- lpicc translates the option to an equivalent newc option
- lpicc ignores the option

The -warn option to the lpicc command line produces information on how lpicc processes each specified cc option. If an option is unknown to lpicc, then an error occurs and the option should not be used. Refer to your UNIX documentation for descriptions of the cc options.

lpicc automatically invokes newc for each specified file having a .c suffix in its filename. Likewise, it invokes the system assembler for filenames having an .s suffix, and it passes all .o files, .a files, and all specified files with unknown suffixes to ldc. Note that ldc is not automatically invoked if you have used the -c option.

There are two valid formats for the lpicc command:

lpicc *source_filename* [*source_filename*] ... [*option*] ...

or

lpicc [*option*] ... *source_filename* [*source_filename*] ...

whereas there is one valid format for the newc command:

newc [*option*] ... *source_filename* [*option*] ...

where,

option represents the compiler options. See the "Compiler Options List" section for the name and description of each LPI option. With lpicc, most compiler options can be placed either before or after *source_filename*. However, an error will result if the -l or the -o option appears before *source_filename*.

source_filename represents the name of the source file. By convention, C source programs usually have .c as a suffix to their filenames.

Note that error messages are directed to the stdout file instead of stderr.

LPI-C Compiler Implementation Limits

The compiler generates error messages if your source program exceeds certain LPI-C compiler implementation limits. Table 1-1 lists these error messages and the corresponding implementation limits.

TABLE 1-1 Compiler Implementation Limits

MESSAGE	LIMIT
Length of line (physical or logical)	No limit
Length of token	256 characters
Length of identifier	256 characters
Length of a string constant (after concatenation)	32767
Number of errors	100
`#include` nesting	No limit
Conditional inclusion nesting	No limit
Number of parameters in macro function definition	256
Number of arguments in macro function invocation	256
Number of macro defintion	No limit
Number of arguments in function call	63
Number of (pointer, array, function) declarator modifiers	15

Compiler Options List

The following list names and describes each of the compiler options and its default setting, where applicable. Compiling a program using the default settings enables the compiler to generate the most efficient code.

OPTION	EXPLANATION
-deb	Produces debugging information for CodeWatch, the LPI source-level debugger. This option must be selected if you intend to use CodeWatch. The default is -nodeb. The -deb option automatically sets the optimization level to 2. When using -deb, you can override the level 2 default by including -opt 1, which sets optimization to level 1, or -noopt, which turns off optimization. You cannot set the optimization level to 3 when using -deb.
-define *name*[=*text*]	Defines *name*, where *name* is a reserved symbol that is predefined by the preprocessor, as if by a **#define** directive. For example, -define VERSION=5 is equivalent to **#define** VERSION 5. If no *string* is given, *name* is defined as 1. Note that no spaces are allowed between *name*, the equal sign (=), and *text*. When using lpicc you can define a function-line macro with the following command line. -define *name* (*[arg[,arg]...]*) = *text* where *arg* is an argument and *text* is the body of the name definition.
-E	Runs the source file through the preprocessor phase only. The output is sent to standard output. The output of -E is suitable for compilation and may be redirected to a file for later use.
-e	Same as the preceeding -E option.
-exp	Produces an expanded listing, including assembly language statements, at the end of the listing file. See the section "The -exp Option" for more information.

OPTION	EXPLANATION
-f387	Generates Intel 80387 floating point instructions where possible. See also the description of the -nof387 compiler option.
-files	Runs the source file through the preprocessing phase only and prints an indented list of all the include files referenced.
-include *file*	Includes (as if by the `#include` directive), the indicated file specified before preprocessing begins. No directory searching is done if the specified file does not exist.
-ipath *dir*[:*dir*]...	Changes the algorithm for searching for `#include` files (whose names are surrounded by quotes and do not begin with /) to look in *dir* before looking in the directories in the standard list.
-istring	Generates in-line code for the LPI-C ANSI library string handling functions where possible.
-l [*fn1*]	Produces a compiler listing, where *fn1* is the name of the file to which the listing is output. If *fn1* is not supplied, make sure that -l is placed after the source filename; otherwise an error message appears if you are using newc. The default name of the file containing the listing is *xxx*.list, where *xxx* stands for the source filename up to, but not including, the last "." (if any) in the name. For example, if the source filename is prog.c, the default output is prog.list. A compiler listing is produced by default whenever -exp, -l, -map, or -xref are specified. See the section "The -l Option" for more information.
-macros	Dumps all of the macro definitions at the end of the preprocessing phase of compilation to the standard output.

OPTION	EXPLANATION
-map	Produces an LPI-C storage allocation map at the end of the listing file. See the section "The -map Option" for more information.
-nof387	Only Intel 80287 floating point instructions are generated. This is the default. See also the description of the -f387 compiler option.
-noobj	Does not produce an object file. Compiles for syntax and semantic checking only. The default is -o.
-noopt	Disables the optimization phase of the C compiler. See the "Optimization" section for more information.
-nowarn	Suppresses level 0 (Note) or level 1 (Warning) error messages. lpicc automatically sets this option when invoking newc. See Appendix B, "Interpreting LPI-C Compiler Error Messages," for more information.
-o *filename*	Names the object file *filename*. If -o is not specified, the default name of the file is *zzz*.o, where *zzz* stands for the source filename up to, but not including, the last "." (if any) in the name. For example, if the source filename is prog.c, then the default object filename is prog.o.
-opt [*lvl*]	Invokes the optimization phase of the LPI-C compiler, where *lvl* stands for the level (1-3) of the optimization. The default is -opt 3. See the "Optimization" section for more information.
-predef	Causes the compiler to print out predefined system names.

OPTION	EXPLANATION
-stat	Prints statistics on the standard output as each compiler phase runs. See the "Compilation Statistics" section for more information.
-stddef	Causes the compiler to predefine system macro names. The -syspath option invokes this option automatically.
-stdpath *dirname*	Changes the default header directory from the default /usr/include/LPI to *dirname*.
-sys	Same as -syspath.
-syspath	Changes the default header directory from /usr/include/LPI to the standard system header directory. This option may be used as a shorthand equivalent to "-stdpath /usr/include," except that this option automatically invokes -stddef.
-undef *name*	Removes any initial definition of *name*, where *name* is a reserved symbol that is predefined by the preprocessor.
-warn	Does not suppress level 0 (Note) messages. This is the default for newc.
-xref	Produces cross-reference information in the listing file. See the description of the -l option for more information.

Compatibility Options

For compatibility with older pre-ANSI C implementations (most notably PCC based compilers), the following compatibility options are supported.

OPTION	EXPLANATION
-xall	Causes **all** of the following compatibility options **except** -xnc, -xws, and -xwc to be turned on.
-xbf	Additionally allows the integral types `char`, `short`, and `long` (signed or unsigned), to be declared as bit-fields, with appropriate maximum widths, without a warning.
-xc	Causes the old-style C formal macro parameter substitution within character constants to be performed. If the -xwc option is also specified, then a diagnostic note will be emitted at each point where this is done.
-xcp	Causes the type `char *` to be accepted as equivalent to the generic pointer type `void *` without a warning.
-xea	Allows the expansion of function-like macro invocations with empty arguments as if the (empty) arguments consisted of no tokens, without a warning.
-xes	Causes the old-style C (pre-ANSI C implementation) file scoping rules to be used for functions and data objects declared as `extern` with an inner block scope.
-xic	Causes types to be assigned to integer constants by old-style C rules rather than ANSI C rules. Refer to Appendix D for a summary of the integer constant typing rules for old-style C and ANSI C.
-xid	Causes (non-ANSI) `#indent` preprocessing directives to be completely ignored without warning.
-xlf	Causes `long float` to be accepted as a synonym for `double` in a declarator.
-xmi	Allows innocuous redefinitions of function-like macros which differ only in the spelling of formal macro parameter identifiers, without a warning.
-xmp	Allows the pernicious (destructive) redefinitions of macros, with a warning.

OPTION	EXPLANATION
-xmr	Allows macros to be redefined by turning on both the -xmp and the -xmi options described earlier in this section.
-xnc	Causes _STDC_ to be undefined (intending to indicate a non-conforming ANSI C implementation).
-xnt	Causes trigraph sequences to be ignored (That is, no trigraph sequence mapping will be performed).
-xoe	Causes only old-style C escape sequences to be recognized within string literals and character constants (that is, the `\a`, `\v`, `\x` *hexadecimal-digits*, and `\?` escape sequences will not be recognized). When any unrecognized escape sequences are encountered, a warning will be given, and compilation will continue as if the backslash in the escape sequence was not present.
-xpg	Causes a warning to be emitted for each unrecognized `#pragma`; by default, the unrecognized `#pragma` are completely ignored.
-xs	Causes old-style C formal macro parameter substitution within string literals to be performed. If the -xws option is also specified, then a diagnostic note will be emitted at each point where this is done.
-xtt	Allows trailing text on a `#else` or `#endif` preprocessing directive, without a warning.
-xup	Causes the integral promotions of integral types in expressions to be applied according to the old-style C "unsigned-preserving" rather than ANSI C "value-preserving" rules. Refer to Appendix D for a summary of the integral promotions for old-style C and ANSI C.
-xwc	Causes warnings to be emitted wherever a function-like macro is defined such that old-style C formal macro parameter substitution within character constants could be performed.

OPTION	EXPLANATION
-xws	Causes warnings to be emitted wherever a function-like macro is defined such that old-style C formal macro parameter substitution within string literals *could* be performed.

Temporary Files

Compiler-generated temporary files are normally placed in the /tmp directory. Use the following command to change the directory where compiler-generated temporary files are placed:

BOURNE SHELL USERS	C-SHELL USERS
TMPDIR=pathname; export TMPDIR	setenv TMPDIR pathname
TMPDIR=. ; export TMPDIR	setenv TMPDIR .

The second form places the temporary files in the current working directory.

Optimization

Programs are optimized at level 3 unless you change the optimization level at compile time. You can do this by specifying one of the following options:

- the -noopt option, which turns off optimization
- a lower optimization level (-opt 1 or -opt 2)
- the -deb option, which sets optimization to level 2

A brief description of the optimizations performed at levels 1, 2, and 3 follows.

LEVEL	DESCRIPTION
1	Operator pattern replacement and Boolean conditional expression optimizations.

LEVEL	DESCRIPTION
2	Common subexpression elimination; all level 1 optimizations.
3	Branch chaining; dead code elimination; flow analysis; loop induction; loop invariant code motion; and all level 1 and level 2 optimizations.

Compiler Listings

The compiler listing options are -l (list), -map (storage allocation map), -xref (cross-reference), and -exp (expanded code). All listing information is directed to the listing file; when any of the listing options are specified, the -l option is invoked by default.

The -l Option

The -l option produces a listing file, which is named by default with the source program name (up to but not including the last ".") and the suffix .list. You can change the name of the listing file by specifying a name after the -l option, in this format: -l *filename*.

The listing file contains a program listing, which consists of a copy of the program with line numbers beginning at 1 in the leftmost column. Included source files are also printed, unless the -noincludes option, which suppresses this output, is specified.

The name of the source file and include file(s) both precede and follow the source text of each file in the listing. Information is provided in three columns preceding each source line, as follows:

- Conditional exclusion of source text is indicated by an x.
- Continuing comment lines are preceded by an asterisk.
- The include nesting level is indicated in brackets.
- The line number in the containing source file is indicated to the left of the source line.

For example,

```
-------------------------------------
MAIN-FILE: "example.c"
-------------------------------------
     1: main ()
     2: {
     3:      /* This is the first line
*    4:         of a continuing
*    5:         comment.
*    6:      */
     7:
     8: #include "hello.h"
-------------------------------------
INCLUDE-FILE: "hello.h"
-------------------------------------
  [1]    1: #if 1
  [1]    2:     printf ("Hello World");
  [1]    3: #else
x [1]    4:     This is excluded text.
x [1]    5: #endif
-------------------------------------
END-INCLUDE-FILE: "hello.h"
-------------------------------------
     9:
    10: }
-------------------------------------
END-MAIN-FILE: "example.c"
-------------------------------------
```

The -map Option

The -map option produces a storage allocation map, which displays the location and allocation size of each program entity. The map categorizes this information by name, class, size, location, and attributes.

This information is presented in five columns:

NAME — Indicates the name of each variable, function, argument, and other program entities.

CLASS — Indicates the storage class of each program entity; for example, whether a variable is static, register, extern, or automatic.

SIZE — Where applicable, indicates the size of the entity, in bytes. Does not indicate the size of externally-defined entities.

LOCATION — Gives the location of the entity in hexadecimal notation.

- For automatic variables, the location column gives the memory offset from the frame pointer.
- For static variables, it gives the memory offset from the beginning of static data.
- For members of structures, it gives the memory offset from the beginning of the containing structure.
- For enumeration constants, it shows the internal value assigned by the compiler.
- For register variables, it shows the name of a register.

The location column does not list locations for labels and external functions and variables, since these are not known until the program is linked.

ATTRIBUTES — Provides more information on some program entities; for example, for a variable, indicates its type.

The map lists external functions and variables first. These externals are listed in order of definition, with name, class, and attributes. A d in brackets, (<d>) after the word "extern" in the class column indicates a defining instance of the external procedure or variable, rather than a reference to an external variable or procedure defined elsewhere.

The map then lists functions and their entities within the program, by line number.

The -xref Option

The -xref option provides cross-reference information for each program entity: the line number on which the entity was declared and the line numbers of all references to it, in the following format:

Declared on *line_number*
[*line_number line_number* ...]

An equals sign (=) following a line number indicates a program line that changes the value of the entity via an assignment statement.

The -xref listing for a program is added to the map listing; when you specify the -xref option, you receive a data storage allocation map by default. The cross-reference information appears on the map in the ATTRIBUTES column for each entity, after the line of map attributes information.

If the program entity is defined in an included file, both the included filename and line number references are given. If references occur in a file other than the included file (such as the source file), that filename is also given. If the program entity is defined in the source file itself, no filename is given.

A sample map listing with the -xref option follows.

```
EXTERNAL ENTRY POINTS

NAME          CLASS         SIZE    LOCATION   ATTRIBUTES

main          extern <d>                       int ()
                                               Declared on 3

Ident1        <constant>              0        enum {}
                                               Declared on 20
                                               21 23 35 46

Ident2        <constant>              1        enum{}
                                               Declared on 20
                                               24 26 37 49

ArrayDim      typedef      10404               int [51][51]
                                               Declared on 70
                                               74 80

printf        extern                           int ()
                                               Declared on 75
                                               75 77

PROCEDURE main ON LINE 3
NAME          CLASS         SIZE    LOCATION    ATTRIBUTES

lower         auto          4       00000014    int =
                                                Declared on 5
                                                include.h
                                                5=
                                                mainprog.c
                                                10= 14

upper         auto          4       00000018    int
                                                Declared on 5
                                                11= 15

step          auto          8       0000001C    double
                                                Declared on 6
                                                12= 19

newnumb       auto          2       00000024    short int
                                                Declared on 7
                                                17 18=

fd_set        <tag>         4       00000032    struct fd_set
                                                Declared on 30
                                                45= 57= 65= 72=
 fds_bits1    <member>      4       00000000    long int
                                                Declared on 32
                                                57 72
 fds_bits2    <member>      4       00000004    long int
                                                Declared on 34
                                                45 57 65
```

The -exp Option

The -exp option produces an expanded code listing for each program, providing the location and low-level language translation for each instruction and label in the program. Each program statement is referenced by line number, with the following information displayed below in four columns.

- The first column displays the offset location of each statement.
- The second column displays the hexadecimal opcode of each instruction.
- The third column displays the instructions in pseudo-assembly code dialect.
- The fourth column displays information about the operands used by each instruction.

Compilation Statistics

The -stat compiler option provides a listing of statistics for each phase of the compile. The statistics are displayed at the terminal by default or can be redirected to a file. The following figure depicts a sample terminal display.

```
PHASE        DISK  SECONDS  SPACE  SYSTEM     CPU

LEX            16        8      5     .25    6.12  16:10:22
ED PARSE        2        0      5     .05     .03  16:10:22
DD PARSE        4        3     76     .07    3.45  16:10 25
PD PARSE       27       13    119     .05   13.85  16:10:38
OPTIMIZER      35        4    119     .12    3.42  16:10:42
ALLOCATOR       0        1    124     .08     .91  16:10:43
PASS3         208       21    160    1.90   16.42  16:11:04
TOTAL         270       39    160    2.15   34.60  16:11:04


Code size      6498
Static size    3152
Source lines    596
Lines/minute    715
```

FIGURE 1-1 Compilation Statistics

The following list names and describes each of the columns displayed in the printout.

- PHASE defines the various phases of the compile. Each phase and its corresponding statistics are displayed upon completion.
- DISK defines the number of intermediate and error message file reads and writes completed during the corresponding phase.
- SECONDS defines the number of seconds elapsed during the corresponding phase.
- SPACE defines the highest page number of the symbol table temporary file used by the compiler during the corresponding phase.
- SYSTEM defines the CPU time used by the operating system while executing the corresponding phase.
- CPU defines the CPU time used by the compiler while executing the corresponding phase.
- The final column provides the current time of day in 24-hour notation.

Linking a Program

To link an LPI-C program, use either the lpild or the ldc command.

- lpild references the latest version of the LPI components installed on your system.
- ldc references the components that were installed with this version of LPI-C.

Refer to Appendix A for details on LPI products and components.

Using lpild

lpild generates executable files (a.out files) from relocatable object files. The lpild command line syntax is as follows:

lpild [*option*] ... *object-file* [*object-file*] ... [*optional-libraries*]

where,

lpild is the command that invokes the linker. *option* represents any of the options described in the "Link Options List" section later in this chapter, or any appropriate UNIX linker option. See the description of the ld command in your UNIX documentation.

object-file specifies object files that are to be linked. Program execution begins at the entry point named `main`, if one exists. Otherwise, execution begins at the first function in the first object file specified on the link command line.

optional-libraries represent file pathnames for libraries that are required for your special application. The standard libraries, libc.a, for example, are always included in the link.

Using ldc

lpild and ldc both generate executable files (a.out files) from relocatable object files. ldc references components, such as libraries, that were installed specifically with LPI-C. On systems with more than one LPI product, lpild references the most recent revision of each component.

The ldc command line syntax is as follows:

ldc [*option*] ... *object-file* [*object-file*] ... [*optional-libraries*]

See the "Using lpild" and "Link Options List" sections for a description of the command options that are acceptable to both lpild and ldc.

Link Options List

The lpild and ldc commands accept any options that are valid for the UNIX ld command. For an explanation of these options, see the description of the ld command in your UNIX documentation. In addition, the following LPI options are acceptable to ldc. (Note that the -verbose option is also accepted by lpild.)

OPTION	EXPLANATION
-sys	Same as -syslib.
-syslib	Specifies that the standard system library (that is, libc.2) will be used to link the program, rather than the LPI-C ANSI library. This option is the default for lpild.
-verbose	Echoes the link line used by lpild to link the program.

Using LPI-C ANSI Libraries vs. System Libraries

When linking C programs, either the LPI-C ANSI library or the system libraries may be linked with object modules to produce an executable program.

When using ldc, the default library to be loaded is the LPI-C library, which is fully ANSI-conformant. ldc is typically used to link object files produced by LPI-C, the system assembler, and possibly other C compilers. The LPI-C library may be deselected by using the -syslib option, which will cause the System libraries (such as libc.a) to be linked in place of the LPI-C library. Note that the system libraries may not be ANSI conformant.

lpild is typically used to produce multi-language programs generated from various LPI compilers. In this case, the system libraries will always be loaded rather than the LPI-C ANSI library.

When linking with the LPI-C ANSI library, all source files should have been compiled in a consistent manner by using the header files residing in the LPI header directory /usr/include/LPI.

Conversely, when -syslib is used to deselect the LPI-C library, all object files to be linked should have been compiled in a consistent manner by using the system include files. The -syspath and -stdpath compiler options change the default `#include` path options from the LPI-C header directory to the system header directory, or to an alternate directory, if desired. See descriptions of these options for more information.

When using either lpild or ldc, attempting to mix the LPI-C library with the system libraries (particularly libc.a) may produce unexpected results at both link time and execution time.

Running a Program

The user program is run by invoking the executable object module produced by the lpild or ldc command. This is done by giving the name of the executable object module as a command. The name of the executable module generated by lpild or ldc is normally a.out unless the -o *filename* option was specified in the link command.

CodeWatch

CodeWatch is a source-level, interactive debugger that enables you to debug LPI-C programs. CodeWatch enables you to control program execution to set breakpoints, monitor what is happening, modify values, and evaluate results. CodeWatch keeps track of variables, subprograms, subroutines, and data types in terms of the symbols used in the source language. You can use this debugger to access the source text of the program, to identify and reference program entities, and to detect errors in the program's logic.

Note
Your programs must be compiled using the -deb option before they can be run under the control of the debugger. When called with the -deb option, the compiler generates a separate symbol table file, *filename*.stb, in the current directory. *filename*.stb contains symbolic information that the debugger uses to reference and manipulate source program symbols and entities, set breakpoints and tracepoints, and control program execution.

For detailed information on CodeWatch, refer to the *CodeWatch Reference Manual.*

Other Debugging Options

A more primitive technique for debugging C programs involves using the UNIX adb or sdb debugger. For a description of the adb or sdb command and options, refer to your UNIX documentation.

Chapter 2: LPI-C Language Specifics

Chapter 2: LPI-C Language Specifics

This chapter describes the LPI-C language implementation for your system.

Data Types

Table 2-1 lists the data types supported by this implementation along with the alignment and size of their machine representations. Subsequent sections provide additional information about the internal representation, size, alignment, and range, where applicable.

TABLE 2-1 Data Types and Sizes

DATA TYPE	ALIGNMENT	SIZE
pointer (*type* *)	32-bit word	4 bytes
`char`	8-bit byte	1 bytes
`unsigned char`	8-bit byte	1 bytes
`double`	32-bit word	8 bytes
`long double`	32-bit word	8 bytes
`enum`	32-bit word	4 bytes
`float`	32-bit word	4 bytes
`int`	32-bit word	4 bytes
`unsigned int`	32-bit word	4 bytes
`long`	32-bit word	4 bytes
`unsigned long`	32-bit word	4 bytes
`short`	16-bit word	2 bytes
`unsigned short`	16-bit word	2 bytes

In addition to these types, the following combinations of type specifiers are allowed and have the meaning shown in Table 2-2.

TABLE 2-2 Type Specifiers

TYPE	MEANING

`signed short`
`short int`
`signed short int`
	`short`

`unsigned short int`	`unsigned short`

`signed`
`signed int`
no type specifiers
	`int`

`unsigned`	`unsigned int`

`signed long`
`long int`
`signed long int`
	`long`

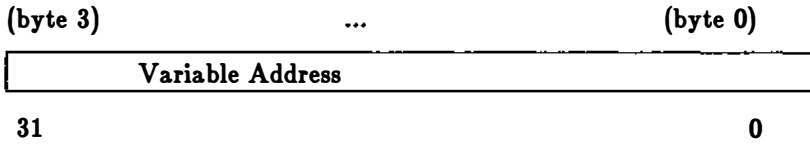
`unsigned long int`	`unsigned long`

Note
The code generated to support the data type *double* as the value returned from a function may not be compatible with other C compilers. If you use this feature, be sure that all modules in the program are compiled using LPI-C.

The size of an aggregate (array, struct, or union) is rounded up to an integral multiple of the aggregate's boundary requirement. See the "Data Type Boundaries Within Structures" section later in this chapter for more information on boundary requirements.

pointer (type *)

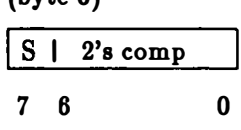
pointer (*type* *) is a 4-byte, 32-bit word variable capable of holding the address of any variable.



Size	Alignment
4 bytes	32-bit word

char

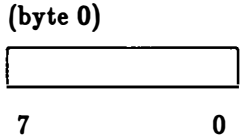
char is an 8-bit, signed, 2's complement binary integer. Bits 0-6 contain the integer, bit 7 contains the sign. The sign bit (S) is 1 if the value is negative, 0 if the value is positive or zero.



Size	Alignment	Range
1 byte	8-bit byte	−128 to 127

unsigned char

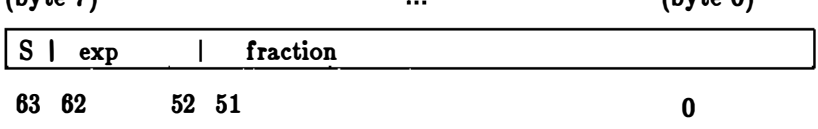
unsigned char is an 8-bit, unsigned, binary integer. Bits 0-7 contain the integer.



Size	Alignment	Range
1 byte	8-bit byte	0 to 255

double

double is an IEEE draft standard, double precision, basic format, binary floating point number. It consists of a 1-bit sign, 11-bit biased exponent (bias = 1023), and 52+(1)-bit binary fraction (hidden bit). An exponent of 2047 represents +/− infinity or not-a-number. +/− infinity is represented by an exponent of 2047 and zero as the fraction part; the sign bit denotes the + or − infinity. Not-a-number is represented by an exponent of 2047 and any non-zero value in the fraction part; the sign bit of not-a-number is not significant. An exponent of 0 denotes a denormalized small value of reduced precision. Both +0 and −0 are possible. Note that the range is approximate and indicates representable values which include zero and both negative and positive numbers.



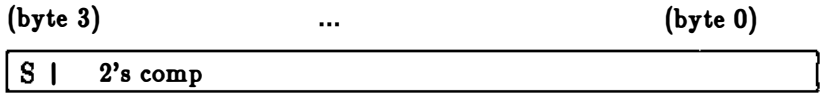
Size	Alignment	Approximate Absolute Value of Range
8 bytes	32-bit word	4.94E-324 to 8.99E+307

long double

long double is implemented as **double**.

enum

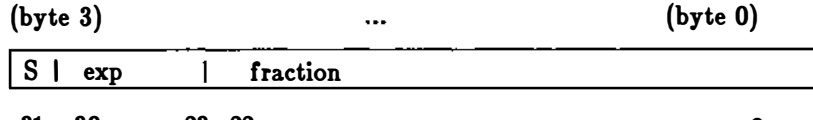
enum has the same representation as **int**; it is a 32-bit, signed, 2's complement binary integer. Bits 0-30 contain the integer, bit 31 contains the sign. The sign bit (S) is 1 if the value represented is negative. See the "Enumeration Types" section later in this chapter for further information.



Size	Alignment	Range
4 bytes	32-bit word	−2,147,483,648 to 2,147,483,647

float

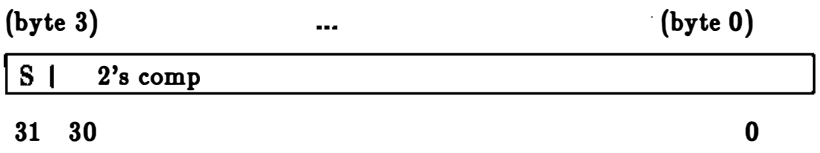
`float` is an IEEE draft standard, single precision, basic format, binary floating point number. It contains a 1-bit sign, 8-bit biased exponent (bias = 127), and 23+(1)-bit binary fraction (hidden bit). An exponent of 255 represents +/− infinity or not-a-number. +/− infinity is represented by an exponent of 255 and zero as the fraction part; the sign bit denotes the + or − infinity. Not-a-number is represented by an exponent of 255 and any non-zero value in the fraction part; the sign bit of not-a-number is not significant. An exponent of 0 denotes a denormalized small value of reduced precision. Both +0 and −0 are possible. Note that the range is approximate and indicates representable values which include zero and both negative and positive numbers.



Size	Alignment	Approximate Range of Values
4 bytes	32-bit word	1.40E-45 to 3.40E+38

int and long

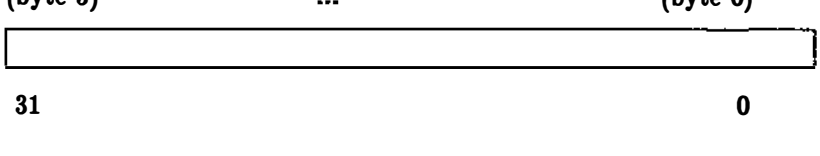
`int` and `long` are 32-bit, signed, 2's complement binary integers. Bits 0-30 contain the integer, bit 31 contains the sign. The sign bit (S) is 1 if the value represented is negative.



Size	Alignment	Range of Values
4 bytes	32-bit word	−2,147,483,648 to 2,147,483,647

unsigned int and unsigned long

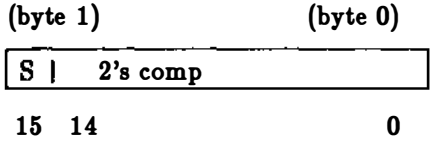
`unsigned int` and **`unsigned long`** are 32-bit, unsigned, binary integers. Bits 0-31 contain the integer.



Size	Alignment	Range of Values
4 bytes	32-bit word	0 to 4,292,967,293

short

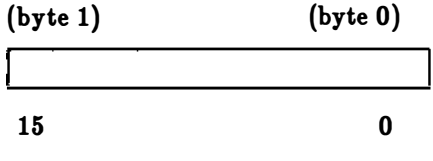
`short` is a 16-bit, signed, 2's complement binary integer. Bits 0-14 contain the integer, bit 15 contains the sign. The sign bit (S) is 1 if the value is negative, 0 if the value is positive or zero.



Size	Alignment	Range of Values
2 bytes	16-bit word	-32768 to 32767

unsigned short

`unsigned short` is a 16-bit, unsigned, binary integer. Bits 0-15 contain the integer.



Size	Alignment	Range of Values
2 bytes	16-bit word	0 to 65535

Implementation Specifics

This section describes elements of LPI-C that may differ from other implementations of C. For a full description of implementation-defined features, see Appendix C.

Length of Identifiers

LPI-C allows identifiers up to 256 characters long.

Integer Constants

Integer constants with values that cannot be represented in 32-bit (4-byte) 2's complement format are not supported.

Character Constants

A character constant of the form *ddd* is sign extended, where *ddd* stands for octal digits representing the value of the character. For example, the character represented by '\377' is equivalent to −1. This implementation supports multi-character character constants. The characters right-to-left in such a constant are assigned to integers from the least significant byte to the most significant byte.

Sign Extension of char Type

Conversions of (signed) `char` type to integer type will sign-extend. The range of values representable by the type `char` is −128 to 127. The ASCII character set is represented by the integers in the range of 0 to 127.

Bit-fields in Structures

Only an integer type can be declared as a bit-field in a structure. The value of a bit-field is treated as an unsigned data item in computations regardless of the actual type specified in the declaration.

Bit-fields are packed into 4-byte integers, which are aligned on 2-byte boundaries and are packed left-to-right. Unused bits occupy the least significant positions in the 4-byte integers.

Right Shift Operator

The right shift operator performs a logical shift if the left operand is unsigned; otherwise, an arithmetic shift is performed. In a logical shift, the high-order bits vacated by the shift are filled with zero-bits. An arithmetic shift fills vacated bits with a copy of the sign bit (1 if negative, 0 otherwise.)

Truncation on Integer Divide

The sign of the result of an integer divide operation is determined by the rules of algebra. The fractional part of a remainder is truncated, and the result of the truncation is always toward zero. For example, −5/2 yields −2.

Mod Operator With Negatives

The result of mod operator x % y is defined to be x−y(x/y), where x/y is the quotient of an integer divide with any fractional part truncated.

Pointer Casts

Pointer casts that specify a change from a pointer to one type into a pointer to another type do not actually change the representation. In the following example:

```
int   i,   *ip;
char  c,   *cp;

ip     =    &i;
cp     =   (char   *)   ip;
```

the value of the pointer `cp` is the address of the most significant byte of the variable `i`.

Directories Searched for #include

The following list explains the method for locating includable source files. An include preprocessing directive of the form:

```
#include <name>
```

specifies that *name* is a source file that will be searched for in the following manner:

- First, if *name* is a fully qualified path name (that is, it begins with a / character), then it will be searched for only in the specified place which begins at the root directory.
- Otherwise (if not found and not fully qualified), then *name* will be searched for within each directory name specified on the command line (in order from left to right) by way of the -ipath compiler option.
- Otherwise (if not found), then *name* will be searched for within the LPI-C standard header directory /usr/include/LPI. This directory can be changed either to the standard system directory /usr/include via the -syspath compiler option, or to any other directory via the -stdpath compiler option.

An include preprocessing directive of the form:

```
#include "name"
```

specifies that *name* is a source file that will be searched for in the following manner:

- First, if *name* is a fully qualified path name (that is, it begins with a / character), then it will be searched for only in the specified place which begins at the root directory.
- Otherwise (if not found and not fully qualified), then *name* will be searched for in the same directory in which the including source file resides.
- Otherwise (if the file is not found), then *name* will be searched for exactly as if it had been included via the first method (that is, `#include <name>`).

Data Type Boundaries Within Structures

Within a structure, each (non-field) member is allocated at a byte offset which is an integral multiple of the boundary requirement for its data type. The boundary requirement for each elementary data type is summarized earlier in Table 2-1. The boundary requirement for an aggregate (array, struct, or union) is defined to be the most stringent boundary requirement of any of its members.

External Procedures

A program can be written in assembler or any available programming language as long as the program observes the same calling conventions as LPI-C. All LPI languages meet this requirement. Also, except where noted in the following sections, the UNIX C compiler, cc, meets these requirements.

Programs compiled by LPI-C can be linked with object modules produced by other languages and the resulting object program will run if the following conditions are met:

- The calling conventions used by other languages must be compatible with the conventions of LPI-C. Other LPI products support these conventions.
- Declarations for formal and actual parameters in the calling and called programs must be written in their respective languages so that the data representations assumed in the declarations agree with each other.

External Names

Certain rules apply with regard to name length and the use of lowercase letters when specifying a name.

Name Length

LPI-C allows names up to 256 characters in length. Some linkers have different length limitations.

Lowercase Names

LPI-C preserves the case of all external names.

Calling Conventions

External procedures are called using the CALL instruction and are returned using the RET instruction. All LPI language compilers use the following calling conventions and are compatible with LPI-C. See the "C Conventions" section at the end of this chapter for additional details concerning the use of LPI-C with other languages.

- The calling procedure pushes the actual arguments onto the stack in the opposite order of their appearance in an argument list. In other words, the last argument is pushed first and the first argument is pushed last.
- A call to a function returning one of the data types that requires a return temporary must load register edx with the address of the temporary.
- The external procedure is called by executing a CALL instruction.

Register Save Conventions

The called procedure is expected to save and restore values in registers esi, edi, and ebx. The calling program can assume that all registers are preserved across the call except for eax, ecx, edx, and the flag registers. The function result is in eax or the result temporary provided by the caller depending on the result data type. See the following "Function Result Conventions" section for more information.

A called procedure is responsible for establishing a new stack frame for itself and saving and restoring all registers except eax, ecx, edx, and the flag registers. The procedure establishes the stack frame by following the following sequence:

```
pushl  ebp
movl   esp, ebp
subl   #framesize, esp
pushl  ebx
pushl  esi
pushl  edi
```

A return must store any function result value, restore registers modified (other than eax, ecx, edx, and the flags register), pop its stack frame, and

return to the caller. The return accomplishes this by a load of eax (or a store to the result temporary), restoring the values of ebx, esi, edi, LEAVE and RET.

Function Result Conventions

A function procedure returns its result value to the calling procedure in one of three ways, depending upon the data type: in register eax, on top of the floating point coprocessor stack, or in the temporary storage area provided by the caller.

Returns in Register eax

The following lists the LPI-C data types returned in register eax.

```
int
unsigned int
long
unsigned long
short
unsigned short
char
unsigned char
pointer to ...
```

Returns in Temporary Storage Area

The struct data type is returned in a temporary storage area. The calling procedure provides the temporary area and loads edx with its address prior to the call.

Returns on Top of Floating Point Coprocessor Stack

The `float` and `double` data types are returned on top of the floating point coprocessor stack.

C Conventions

All LPI language compilers use the calling conventions previously described, but there are two treatments of arguments by LPI-C that require special attention. The first is that C functions expect argument values on the stack, whereas other languages expect the stack to contain pointers to the arguments. The second is that the declaration of arguments in C specifies the data type of the value on the stack.

When interfacing with LPI-C or the UNIX C compiler, cc, with other LPI languages, the C function arguments must be declared as ptr to *type*, where *type* is a C data type that is equivalent to the other data type declared in the calling procedure for the corresponding argument. The only exception to the ptr to *type* rule is that character string arguments can be optionally declared in C as array of **char** instead of ptr to **char**. This is almost equivalent and both indicate that the stack contains a ptr to **char** for the corresponding argument.

The UNIX C compiler, cc, also uses the same compatible calling conventions and argument passing as discussed previously for LPI-C, but with the following exception:

A C function that returns a structure result (**struct** {...}) when compiled with the UNIX cc compiler may expect the caller to supply a temporary area for the result. The caller is then expected to push a pointer to this area onto the stack immediately after pushing the arguments.

LPI-C uses a slightly different convention. The caller places the pointer to the result into the edx register prior to the call, rather than pushing it onto the stack. In general, when interfacing to subroutines written in another LPI language, you should use LPI-C rather than the UNIX cc compiler.

LPI Equivalent Data Types

The following table lists compatible data types of all LPI languages for determining equivalent argument type correspondence.

TABLE 2-3 LPI Equivalent Data Types

RPG II	PL/I	COBOL	C	FORTRAN	BASIC	PASCAL
BINARY (4-bytes)	FIXED BIN,p>15	COMP PIC S9(5)-S9(9)	int or long	INTEGER*4	-	INTEGER;
BINARY (2-bytes)	FIXED BIN,p≤15	COMP PIC S9(1)-S9(4)	short	INTEGER*2	INTEGER(%)	INTEGER
-	-	-	char	INTEGER*1	-	-
ALPHA -MERIC	CHARAC -TER(n)	DISPLAY A(n) or PIC X(n)	char [n]	CHARAC -TER*n	-	PACKED ARRAY[1...n] OF CHAR
-	CHARAC -TER(n) VARYING	-	struct {short s; char c[n];}	-	-	-
-	CHARAC -TER(1)	-	char	CHARAC -TER*1	-	CHAR
-	CHARAC -TER(*)#	-	char [ ]#	CHARAC -TER*(*)#	-	-
PACKED DECIMAL	FIXED DECIMAL	COMP-3 PIC S9(n)	-	-	-	-
ZONED DECIMAL	-	DISPLAY PIC S9(n) SIGN TRAILING	-	-	-	-
LEADING SIGN	-	DISPLAY PIC S9(n) SIGN LEADING SEPARATE	-	-	-	-
TRAILING SIGN	-	DISPLAY PIC S9(n) SIGN TRAILING SEPARATE	-	-	-	-
INDICA -TOR	BIT(1)	-	-	-	-	-
-	BIT(1) ALIGNED	-	char	-	-	BOOLEAN
-	BIT(n) ALIGNED	-	-	-	-	SET
-	-	-	int or long	LOGICAL*4	-	-
-	-	-	short	LOGICAL*2	-	-
-	-	-	char	LOGICAL*1 /BYTE	-	-
-	FLOAT BIN,p>23	COMP-2+	double	REAL*8	REAL(#) (MBASIC)	-
-	FLOAT BIN(p),p≤23	COMP-1+	float	REAL*4	REAL(!) (MBASIC)	REAL
-	-	-	struct {float real, imaginary;}	COM -PLEX*8	-	-
-	POINTER	-	ptr to ...	-	-	POINTER
-	LABEL	-	-	alternate return	-	-
-	ENTRY	-	-	Dummy procedure	-	-
-	-	-	-	-	REAL (CBASIC)	-
-	-	-	-	-	STRING(8)++	-

The symbols used in Table 2-3 have the following meaning:

‡ When compiled with the -longint option.

‡‡ Use only when calling from PL/I to C or from FORTRAN to C, and not from C nor between PL/I and FORTRAN.

\+ Standard COBOL representation.

++ Stored as 8-bytes: a 2-byte length, followed by a 4-byte address, followed by a 1-bit flag, followed by 15-bits reserved.

Appendix A: Installing, Listing, and Removing LPI Software

This appendix describes the lpiadmin utility commands and options for installing, updating, listing, checking, and removing LPI software.

The lpiadmin Utility

The lpiadmin utility is a tool that allows you to perform specific administrative tasks. The utility assists you during the initial installation of LPI products, when you are updating LPI software, when you remove LPI software from your system, and during LPI product checks. The lpiadmin utility provides five procedures that simplify these administrative tasks; each procedure is invoked with a specific lpiadmin command.

TABLE A-1 lpiadmin Commands

COMMAND	RESULT
lpiadmin install	Installs your product components. See the "Installing LPI Products" section for the command syntax and information.
lpiadmin help	Lists the lpiadmin program commands and the syntax for each.
lpiadmin list	Provides a listing of all LPI product components currently installed on your system. See the "Listing LPI Products" section for the command syntax and information.
lpiadmin remove	Removes a specified LPI product from your system. See the "Removing LPI Products" section for the command syntax and information.

TABLE A-1 lpiadmin Commands (Cont.)

COMMAND	RESULT
lpiadmin check	Checks the consistency of LPI products. See the "Checking LPI Products" section for the command syntax and information.

After installation, lpiadmin resides by default in the LPI administration directory, /usr/lib/LPI, where it is referenced to invoke the lpiadmin help, lpiadmin list, lpiadmin remove, and lpiadmin check procedures.

Invoking lpiadmin Procedures

There are two methods for invoking the majority of lpiadmin procedures. See the Note within the "Invoking lpiadmin install" section for details on the exception.

1. From the /usr/lib/LPI directory, invoke the procedure by entering the command name and, if desired, the command options. For example:

   lpiadmin list [*option*] ...

2. From another directory, invoke the procedure by entering the full command pathname and, if desired, the command options. For example:

   /usr/lib/LPI/lpiadmin list [*option*] ...

The following sections describe each procedure, its command name and options, and a sample of the terminal display, where relevant.

Installing LPI Products

Your product Release Notes contain a listing of the product components to be installed. The listing includes the name of each product (file), its default directory, and the release number.

Preparation

There are certain procedures that you must follow before installing the product components. See your product Release Notes for these step-by-step instructions. When you complete these procedures, the components reside in a temporary directory on your system, and you can begin the installation procedure.

Comparing lpiadmin Revision Numbers

If you already have LPI products installed at your site, we recommend that you compare the version number of the current lpiadmin utility (the one bound with the current release) with the version number of the lpiadmin utility that resides in your administration directory. By comparing these two numbers, you can determine which is the latest version of the utility. This information is important when upgrading or supplementing your LPI products and components. Using the latest version of lpiadmin ensures that all installed product components, particularly shared components, can be easily accessed by lpiadmin procedures.

Although it is likely that the contents of the release media contain the latest version of lpiadmin along with your new products and components, it is wise to confirm this before beginning the installation. To make the comparison, go to the administration directory and invoke the lpiadmin list procedure, using the following command:

```
lpiadmin list -q -l | grep lpiadmin
```

The lpiadmin utility list displays a quick listing of the lpiadmin versions installed on your system. For example:

```
lpiadmin          01.09.00   L  /usr/lib/LPI/lpiadmin
lpiadmin          01.09.00   L  /usr/lib/LPI/lpiadmin
lpiadmin          01.09.00   L  /usr/lib/LPI/lpiadmin
lpiadmin          01.09.00   L  /usr/lib/LPI/lpiadmin
lpiadmin          01.09.00   L  /usr/lib/LPI/lpiadmin
lpiadmin          01.09.00   L  /usr/lib/LPI/lpiadmin
```

Compare the listed version numbers with the version number specified in your Release Notes.

- If the version that is bound with the release media is the most current, return to the temporary directory which contains the contents of the release media and begin the installation as described in the "Invoking lpiadmin install" section.

- If the version bound with the release media is not the most current, stay in the administration directory and copy the most current version to the temporary directory. To copy lpiadmin from the administration directory, you must log in as super-user (su) or root, be in the administration directory, and use the cp command.

  The following command shows the syntax; you must specify the actual pathname of the temporary directory where the contents of the release media reside:

  cp lpiadmin */temporary_directory_pathname*

  Once the copied version resides in the temporary directory, you can begin the installation.

The lpiadmin install Procedure

By default, the lpiadmin install procedure moves the component files from the temporary directory to a specific destination directory. In general, accepting the default directory satisfies the disk space requirements of most installations. However, if your site has unique requirements, you can tailor the installation procedure to meet these requirements. See the "Installing Software Interactively" section for details.

Invoking lpiadmin install

To invoke the lpiadmin install procedure, you must log in as super-user (su) or root and be in the temporary directory that contains the contents of the release media.

The lpiadmin install command has the following format:

lpiadmin install [-c] [-i] [-v] [-q]

where,

lpiadmin install is the command that invokes the installation procedure.

-c (copy) allows you to copy the files to their destination directory. By default, the installation procedure moves rather than copies files. At the end of the install procedure all product components reside in the destination directory. You can retain a copy of the files in the temporary directory by performing the installation with the -c option.

-i (interactive) allows you to perform the installation interactively. See the "Installing Software Interactively" section for details.

-v (verbose) produces a printout of the lpiadmin activities at each stage of the installation.

-q (quick) suppresses consistency checking of previously installed LPI products. Using this option reduces the amount of time required for the installation. See the "Checking LPI Products" section for more information.

Note
To ensure that the version of lpiadmin residing in the temporary directory is being referenced, precede the command with the pathname ./ as shown in the following example:

```
./lpiadmin install
```

Installing Software Interactively

Using lpiadmin install interactively allows you to tailor the installation to meet the unique needs of your site. To install product components interactively, include the -i option on the lpiadmin install command line. For example:

```
./lpiadmin install -i
```

When you perform the installation interactively, the procedure identifies its activity and, where relevant, prompts you for a response. For example, lpiadmin install identifies components that can be installed at a location of your choice. For such components, the procedure first specifies the default destination and prompts you to specify a non-default destination, if desired. You can accept the default location or specify an alternative location. If you are specifying an alternative, enter the full pathname of that directory. The system confirms your selection, and moves the file to that directory.

Command Examples

The following examples demonstrate how to use the lpiadmin install command and options.

1. To move components from the temporary directory to the default destination directories and get a printout of lpiadmin activity, use the following command:

   ```
   ./lpiadmin install -v
   ```

2. To move components interactively from the temporary directory to the default (or other) destination directories and get a printout of the interaction, use the following command:

   ```
   ./lpiadmin install -i -v
   ```

3. To move components from the temporary directory to the default destination directories, suppress consistency checking of already installed LPI products, and get a printout of lpiadmin activity, use the following command:

   ```
   ./lpiadmin install -q -v
   ```

4. To copy components from the temporary directory to the default destination directories, use the following command:

   ```
   ./lpiadmin install -c
   ```

   The components reside in two locations after this command is executed.

Listing LPI Products

Prior to installation, you used your Release Notes to get a listing of the product components contained on the release media. After installation, you can get a complete listing of all LPI products installed on your system by using the lpiadmin list procedure. The listing includes the product name, version, installation date, and similar information. For example:

```
LPI-COBOL              05.61.00     (9.28.88/12:03:35)
LPI-FORTRAN            03.01.00     (10.3.88/14:47:57)
LPI-PL1                03.05.20     (10.17.88/17:47:56)
LPI-RPG                02.50.00     (12.14.88/13:57:31)
LPI-PASCAL             02.10.00     (7.13.88/13:48:43)
LPI-BASIC              02.05.00     (4.14.87/14:46:24)
LPI-C                  03.01.00     (10.17.88/09:31:16)
CodeWatch              04.05.00     (11.21.88/10:37:31)
```

Invoking lpiadmin list

The lpiadmin list command has the following format:

lpiadmin list [-l] [-q]

where,

lpiadmin list is the command that invokes the listing procedure.

-l (long) produces a listing of all files (components) associated with each product that has been installed to date. If the -q option is not specified, the system performs a consistency check on each component before it displays the listing. This operation may take several minutes before the listing appears on your screen.

-q (quick) suppresses consistency checks on installed LPI products. Since consistency checks are suppressed, the listing appears on your screen shortly after the command is invoked.

Command Examples

The following command examples demonstrate how to use the lpiadmin list command and its options.

1. To get a quick listing of products installed on your system and suppress consistency checking of installed product components, use the following command:

   ```
   lpiadmin list -q
   ```

2. To get a long listing of all components associated with each product installed on your system and to allow consistency checking, use the following command:

   ```
   lpiadmin list -l
   ```

   The previous display shows a sample listing produced by the lpiadmin list -l command. The output from this command is particularly useful when working with the lpiadmin check procedure. You can examine this listing for specific component names.

Checking LPI Products

The lpiadmin check procedure provides a consistency check on component files. The procedure searches for component files, examines them, generates data on particular items, and identifies whether the current version differs from the installed version. By default, lpiadmin check automatically performs a consistency check during the installation, listing, and remove procedures. Note that you can suppress the check by including the -q option in any of the corresponding command lines.

When the lpiadmin check procedure detects an error, it responds as follows:

- If the procedure is operating non-interactively (that is, performing a consistency check during another lpiadmin procedure), it informs you that an error exists and prompts you to invoke lpiadmin check for more information.

- If the procedure is operating interactively (that is, performing a consistency check in response to the lpiadmin check command), it prompts you to decide whether or not you want the inconsistency corrected.

It is particularly useful to check your components when there is a question of file corruption. The procedure checks the files and, when relevant, works interactively with you to correct the problem.

Invoking lpiadmin check

To invoke the lpiadmin check procedure, you must log in as super-user (su) or root.

The lpiadmin check command has the following format:

lpiadmin check [-v] [*component_name*] ...

where,

lpiadmin check is the command that invokes the checking procedure.

-v (verbose) produces a full printout of the lpiadmin activities during the checking procedure.

component_name represents the name of the product component to be checked. There are two types of components. Regular components are files that have a standard filename, such as lpibasic. Indirect components are files that have a shell environment variable name, such as LIB_LPI. You can identify the component type by examining the second field of the lpiadmin list -l output. See the "Command Examples" section for details on specifying each type of component.

By default, the procedure checks all installed components.

Command Examples

1. To perform a consistency check on all installed LPI products, and get a printout of lpiadmin activities, use the following command:

   ```
   lpiadmin check -v
   ```

2. To perform a consistency check on a regular product component and get a printout of lpiadmin activities, use the following command:

   ```
   lpiadmin check -v lpibasic
   ```

   In this case, the consistency check and printout is performed on lpibasic.

3. To perform a consistency check on an indirect product component use the following command:

   ```
   lpiadmin check LIB_LPI
   ```

   Note that in this case, lpiadmin checks all versions of the component identified in the lpiadmin list output by the shell environment variable name LIB_LPI which exist on the system. Multiple versions of indirect components may exist.

Removing LPI Products

The lpiadmin remove procedure allows you to remove an LPI product from your system. An LPI product may have several components. Some of these components are used exclusively by that product. For example, the lpibasic compiler is used exclusively by the LPI-BASIC product. Other components, such as the lpild linker, are used by several products.

The lpiadmin remove procedure does not necessarily remove a component that is used by other products if these other products still exist on the system. Consequently, after you remove a product, certain (shared) components may remain.

There are only two cases when the system removes a shared component associated with a product:

- If the shared component that is associated with the product being removed is not the latest version of that component on the system.
- If the specified product is the last LPI product on your system.

Invoking lpiadmin remove

To invoke the lpiadmin remove procedure, you must log in as super-user (su) or root. The lpiadmin remove command has the following format:

lpiadmin remove [-n] [-v] [-q] *product_name*

where,

lpiadmin remove is the command that invokes the remove procedure.

-n (no action) gives a list of actions which would be performed if the product were to be removed. Nothing is actually removed from your system if you include this option on the command line.

-v (verbose) prints the lpiadmin activities at each stage of the removal procedure.

-q (quick) produces a faster remove procedure by suppressing consistency checking of installed LPI products.

product_name represents the name of the product that you want removed. Use the lpiadmin list command to identify the exact name of the product. The lpiadmin remove procedure is not case-sensitive; you can use either uppercase or lowercase when specifying the product name.

Command Examples

1. Assume that you want to test a situation before actually removing a product. You can get a listing of the lpiadmin activity that would occur if you removed a particular product by using the following command:

   ```
   lpiadmin remove -n lpi-cobol
   ```

   In this case, the listing is produced of the lpiadmin activities that would occur if LPI-COBOL were removed.

2. To remove a product, suppress consistency checking, and get a printout of the lpiadmin activity, use the following command:

   ```
   lpiadmin remove -q -v lpi-pascal
   ```

   In this case, LPI-PASCAL is removed, consistency checking is suppressed, and a printout of lpiadmin activity is generated.

3. To remove a product, allow consistency checking, and get a printout of the lpiadmin activity, use the following command:

   ```
   lpiadmin remove -v LPI-C
   ```

   In this case, the command line contains the product name LPI-C in uppercase letters. The remove procedure is not case-sensitive; you can enter the name in either uppercase or lowercase.

Appendix B: LPI-C Compiler Error Messages

A compilation error message is in the following format:

List-of-tokens-surrounding-the-offending-token-pointed-to-by-a-caret-below

** *phase error* (*code*): `line` *line_number*, "*file_name*"
Description-of-the-error-consisting-of-one-or-more-lines

where,

phase	is the compilation phase in which the error occured. The phases are listed and described in Table B-1.
error	is a brief statement of the error. The error messages are listed and described in Table B-2.
code	is the error code.
line_number	is the physical line number on which the error was encountered (taking into account any `#line` directives which may have been specified).
file_name	is the name of the file in which the error was encountered (taking into account any `#line` directives which may have been specified).

The following is an example of an LPI-C compiler error message.

```
<bof> static int i [ 2 ] = { 1 , 2 , 3 } ; <eof>
                             ^

** SYNTAX ERROR-2 (74): line 1, "example.c"
   Too many initial values for this array.

1 error detected
```

The following table lists and describes the compilation phases.

TABLE B-1 Compilation Phases

PHASE	DESCRIPTION
SCANNER	Indicates that an error was encountered within the lowest lexical scanning/analysis phase of compilation (that is, translation phases 1 through 3). Note that in this case, the list of tokens and the caret will not be printed.
PREPROCESSOR	Indicates that an error was encountered within the preprocessng phase of compilation (that is, translation phase 4). Note that in this case, the list of tokens and the caret will not be printed.
LEXICAL	Indicates that an error was encountered within the phase of compilation which translates character constants, string literals, and converts preprocessing tokens into tokens (that is, translation phases 5, 6, and part of 7).
SYNTAX	Indicates that an error was encountered within the syntactic analysis phase of compilation (that is, translation phase 7).
SEMANTIC	Indicates that an error was encountered within the semantic analysis phase of compilation (that is, translation phase 7).

The following table lists the level, class, and description of the possible compiler error messages.

TABLE B-2 Compiler Error Messages

SEVERITY LEVEL	ERROR CLASS	DESCRIPTION
0	NOTE	Compiler detected a questionable but legal construct; compilation will continue unhindered. The NOTE error message may be suppressed with the -nowarn option. (See the "Compiler Option List" in Chapter 1 for more information.)
1	WARNING	Compiler detected a technically illegal, unusual, or non-portable source language construct; if this was intended, the message can be ignored. WARNING messages can be suppressed by the -nowarn compiler option.
2	ERROR-2	Compiler detected an invalid source language construct which the compiler attempts to ignore and continues with compilation. Errors with this level of severity do not prevent the generation of an object file, but the generated code for statements with errors of this severity is not correct. The source program should be corrected and recompiled.
3	ERROR-3	Compiler detected an invalid source language construct. Compilation of the statement containing the error is abandoned and no compilation can continue. No object file is generated. The source program must be corrected and recompiled.

TABLE B-3 Compiler Error Messages (Cont.)

SEVERITY LEVEL	ERROR CLASS	DESCRIPTION
4	ABORT	Compiler detected an uncorrectable error, and compilation cannot continue; all previously detected ERRORs must be corrected and the source program must be recompiled.

Appendix C: LPI-C Implementation-Defined Behavior

Implementation-Defined Behavior

Implementation-defined behavior (for a correct program construct) depends upon the characteristics of the implementation. Every implementation of ANSI C is required to supply a description of each of these characteristics.

The following is a description of all of the implementation-defined features of ANSI C and (following the "☞" symbol) the corresponding defined behavior for this version of the LPI-C on your system.

1. Translation

i. How a diagnostic is identified:

☞ See Appendix B.

ii. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character that is implementation-defined:

☞ Nonempty sequences of white-space characters will be (conceptually) replaced by one space character (in translation phase 3). Note that the only circumstance in which this is important is within the argument list of the invocation of a function-like macro that uses the preprocessing stringize (#) operator.

2. Environment

i. The semantics of the arguments to `main`:

☞ If defined as

```
main (argc, argv) int argc; char *argv[];
```

- `argc` will set to the total number of command-line arguments with which the program was invoked (including the program name itself).
- `argv` will represent an array of pointers to null-terminated strings such that `argv[0]` represents the name of the program itself, and `argv[1]` thru `argv[argc-1]` represent the `argc` command-line arguments available to the program.

Note that it is not required that `argc` and `argv` be the parameter names.

ii. What constitutes an interactive device:

☞ An interactive device is one for which the system library function `isatty()` returns a non-zero value. (See section ttyname (3) in your UNIX documentation.)

3. Identifiers

i. The number of significant initial characters in an identifier without external linkage (at least 31 guaranteed by any ANSI C implementation):

☞ The first 256 characters of an identifier with internal linkage will be regarded as significant.

ii. The number of significant initial characters in an identifier with external linkage (at least 6 are guaranteed by any ANSI C implementation):

☞ The first 256 characters of an identifier with external linkage will be regarded as significant.

iii. Whether case distinctions are significant in an identifier with external linkage:

☞ Case distinctions are significant.

4. Characters

i. The members of the source and execution character sets, except as explicitly specified in the Standard:

☞ The source and execution character sets are identical, and uses the 7-bit ASCII character set stored in an 8-bit byte. (See section ascii (7) in your UNIX documentation.)

ii. The shift states used for the encoding of multibyte characters:

☞ There are no shift states used in encoding multibyte characters, and there are no recognized multibyte characters.

iii. The value of each escape sequence:

☞ The escape sequence values are defined as follows:

ESCAPE	ASCII ACRO-NYM	ASCII OCTAL VALUE	ASCII DECIMAL VALUE	ASCII HEX VALUE
\a (alert)	BEL	007	7	0x07
\b (backspace)	BS	010	8	0x08
\f (form feed)	FF	014	12	0x12
\n (new line)	LF	012	10	0x0A
\r (carriage return)	CR	015	13	0x0D
\t (horizontal tab)	HT	011	9	0x09
\v (vertical tab)	VT	013	11	0x0B

iv. The number of bits in a character in the execution character set:

☞ There are 8 bits per character in the execution character set.

v. The mapping of members of the source character set (in character constants and string literals) to members of the execution character set:

☞ Each character in the source character set (in character constants and string literals) is mapped to the corresponding character in the ASCII character set.

vi. The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant:

☞ If the value of an integer character constant is less than or equal to UINT_MAX, then the value is that of the least significant byte. Otherwise, the value is unpredictable.

vii. The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character:

☞ The characters (or escape sequence values) in a character constant are stored, one (ASCII) character value per byte, in order from right to left into a 32-bit integer from the least significant to the most significant byte. That is, the value of the right most character in the character constant is stored in the least significant byte of the 32-bit integer, the value of the next character to the left in the character constant is placed in the next least significant, and so on. However, if there are more than 4 characters in the character constant (that is, corresponding to 4 bytes per 32-bit integer), then the right most excess characters will be discarded. If there are fewer than 4 characters in the character constant, then the unused most significant bytes are set will be set to zero. For example, `'abcdef'` is effectively equivalent to `0x61626364`; and `'ab'` is effectively equivalent to `0x00006162`.

viii. The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant:

☞ The only locale supported is the `"C"` locale.

ix. Whether a "plain" `char` has the same range of values as `signed char` or `unsigned char`:

☞ A "plain" `char` has the same range of values as a `signed char`.

5. Integers

i. The representations and sets of values of the various types of integers:

☞ Integers are stored in two's-complement binary format. The ranges for the various integer types are given in the standard include file `<limits.h>`.

ii. The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented:

☞ When an integer is converted to a shorter signed integer, the low order (that is, least significant) bits of the longer integer are used; the most significants bits are ignored. Note that a negative value may result. For example, the result of converting the long (4-byte) integer 0xabcd1234 to a short (2-byte) integer would be 0x1234.

When an unsigned integer is converted to a signed integer of equal size, there is no change in the representation. Note that a negative value may result.

iii. The sign of the remainder on integer division:

☞ The sign of the result of an integer divide is determined by the normal rules of algebra. The fractional part of a remainder is truncated, and the result of the truncation is always toward zero. For example, $-5/2$ would yield -2.

iv. The results of bitwise operations on signed integers:

☞ When used with signed operands, the AND ( ), OR (|), XOR (^), complement ( ), and the left shift (<<) bitwise operators, the sign bit (that is, the most significant bit) is treated simply as any other bit in the integer. See below for the result of the right shift (>>) operation on signed integers.

v. The result of a right shift of a negative-valued signed integral type:

☞ When a signed integral type is right-shifted, vacated bits are filled with a copy of the sign bit (that is, the shift is arithmetic rather than logical).

6. Floating point

i. The representations and sets of values of the various types of floating-point numbers:

☞ Type `float` is represented according to the ANSI/IEEE standard 754-1985 for single precision. Type `double` is represented according to the same standard for double precision. The representation of type `long double` is the same as that of type `double`.

ii. The direction of truncation when an integral number is converted to a floating point number that cannot exactly represent the original value:

☞ Truncation is always toward nearest.

iii. The direction of truncation or rounding when a floating point number is converted to a narrower floating point number:

☞ Truncation is always toward nearest.

7. Arrays and pointers

i. The type of integer required to hold the maximum size of an array; that is, the type of the `sizeof` operator (`size_t`):

☞ The type `size_t` is `unsigned int`.

ii. The result of casting a pointer to an integer or vice versa:

☞ When casting back and forth between pointers and integers (a generally bad/non-portable practice), there is no actual change in the value or representation, since both pointers and integers (of type `int` and `long`) are represented in 32-bit machine integers.

iii. The type of integer required to hold the difference between two pointers to members of the same array (that is, `ptrdiff_t`):

☞ The type `ptrdiff_t` is `int`.

8. Registers

i. The extent to which objects can actually be placed in registers by use of the `register` storage-class specifier:

☞ The register storage class specific is ignored.

9. Structures, unions, enumerations, and bit-fields

i. Whether a member of a union object can be accessed using a member of a different type:

☞ This is strictly forbidden; the compiler will emit an appropriate diagnostic.

ii. The padding and alignment of members of structures (this should present no problem unless binary data written by one implementation are read by another):

☞ Within a structure, each non-bit-field member is allocated at a byte offset (from the beginning of the structure) which is an integral multiple of the boundary requirement for its data type. The boundary requirements for each scalar type were summarized earlier in Table 2-1. The boundary requirement for an aggregate (that is, array, structure, or union) type is defined to be the most stringent boundary requirement of any of its member types.

iii. Whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field:

☞ A "plain" `int` bit-field is treated as an `unsigned int`.

iv. The order of allocation of bit-fields within an `int`:

☞ Bit-fields are packed into 4-byte integers, which are aligned on 2-byte boundaries. The bits are allocated within the bit-field storage unit order starting from the most significant bit toward the least significant bit.

v. Whether a bit-field can straddle a storage-unit boundary:

☞ A bit-field may not straddle a storage-unit (that is, `int`) boundary.

vi. The integer type chosen to represent the values of an enumeration type:

☞ An `int` is used to represent enumeration types.

10. Type qualifiers

i. What constitutes an access to an object that has volatile-qualified type:

☞ Any reference (read or write) to the actual memory location corresponding to a volatile qualified object constitutes an access.

11. Declarators

i. The maximum number of (pointer, array, and function) declarators that may modify an arithmetic, structure, or union type:

☞ Fifteen (this implies that there are 11,352,234 legal basic derived declarator types).

12. Statements

i. The maximum number of `case` values in a `switch` statement:

☞ No limit.

13. Preprocessing directives

i. Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value:

☞ A single-character constant used within the preprocessor expression to control a conditional inclusion (that is, with a `#if` or `#elif` directive) does match the value of the same character

constant in the execution character set. Such a value may have a negative value.

ii. The method for locating includable source files:

☞ For information on locating includable source files, see the "Directories Searched for `#include`" section in Chapter 2.

iii. The support of quoted names for includable source files:

☞ Includable source file-names quoted with a pair of double quotes (") are taken verbatim. Includable source file-names quoted with a left and right angle bracket pair (<>), on the other hand, are created by concatenating the preprocessing tokens between the angle brackets in order (from left to right) while condensing any sequence of white-space to one space.

iv. The mapping of source file character sequences:

☞ Source file characters are mapped to their corresponding ASCII values.

v. The behavior on each recognized `#pragma` directive:

☞ Each `#pragma` directive will be ignored.

vi. The definitions for `__DATE__` and `__TIME__` when, respectively, the date and time of translation are not available:

☞ Not applicable; these are defined for this implementation.

vii. The maximum nesting level for `#include` files:

☞ There is no explicitly imposed limit on the number of nesting levels for `#included` files. This implementation will handle cases in which the maximum number of open files (a restriction imposed by the operating system) has been reached, by closing previously opened `#include` files when necessary.

14. Library functions

i. The null pointer constant to which the macro `NULL` expands:

☞ The null pointer constant macro `NULL` (defined in `<stddef.h>`, `<locale.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, and `<time.h>`) expands to: `((void *)0)`.

ii. The diagnostic printed by and the termination behavior of the `assert` function:

☞ The behavior of the `assert` macro is given in standard include file `assert.h`.

iii. The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` functions:

☞ The function `isalnum` tests for the ASCII characters 'A' through 'Z', 'a' through 'z', and '0' through '9'.

☞ The function `isalpha` tests for the ASCII characters 'A' through 'Z' and 'a' through 'z'.

☞ The function `iscntrl` tests for the ASCII characters with decimal values in the range 0 through 31 and also 127.

☞ The function `islower` tests for the ASCII characters 'a' through 'z'.

☞ The function `isprint` tests for the ASCII characters with decimal values in the range 32 through 126; that is, blank through ~ (tilde).

☞ The function `isupper` tests for the ASCII characters 'A' through 'Z'.

iv. The values returned by the mathematics functions on domain errors:

☞ On domain errors, the mathematical functions return values as follows:

FUNCTION	RETURN VALUE
acos	0.0
asin	0.0
atan2	0.0
cos	0.0
log	−HUGE_VAL
log10	−HUGE_VAL
pow	1.0 if base = 0, else 0.0
sin	0.0
sqrt	0.0

v. Whether the mathematics functions set the integer expression `errno` to the value of the macro **ERANGE** on underflow range errors:

☞ Mathematical functions do not set errno on underflow.

vi. Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero:

☞ If the second argument to function `fmod` is zero, a domain error occurs, and the return value is not meaningful.

vii. The set of signals for the `signal` function:

☞ See the LPI-C header file `< signal.h >`.

viii. The semantics for each signal recognized by the `signal` function:

☞ The set of signals for the `signal` function and their meanings is given in the include file `signal.h`.

ix. The default handling and the handling at program startup for each signal recognized by the `signal` function:

☞ The default handling for all signals is set to SIG_DFL, except SIGFPE, which is set to SIG_IGN. (See your UNIX documentation for more information.)

x. If the equivalent of "`signal(sig, SIG_DFL);`" is not executed prior to the call of a signal handler, the blocking of the signal that is performed:

☞ The equivalent of "`signal (sig, SIG_DFL);`" is performed for all signals except `SIGILL`.

xi. Whether the default handling is reset if the `SIGILL` signal is received by a handler specified to the `signal` function:

☞ No, default handling is not reset in `SIGILL`.

xii. Whether the last line of a text stream requires a terminating new-line character:

☞ The last line of a text stream read by Standard I/O need not end in a new-line.

xiii. Whether space characters that are written out to a text stream immediately before a new-line character appear when read in:

☞ All characters written to a text stream will appear when the stream is read.

xiv. The number of null characters that may be appended to data written to a binary stream:

☞ No nulls are necessarily appended to a binary stream.

xv. Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file:

☞ When a stream is opened using any append mode, the file position indicator is initially at the end of the file.

xvi. Whether a write on a text stream causes the associated file to be truncated beyond that point:

☞ Writing to a text stream does not cause truncation of the file.

xvii. The characteristics of file buffering:

☞ Fully buffered files: Input characters are read a block at a time and doled out upon request. Output characters are accumulated in a block and flushed (written out) when: the block is full, `fflush` is called, or input is requested on a stream that is not fully buffered.

Line buffered files: On input, characters are read until a new-line is found, upon which the entire line is made available. On output, characters are buffered in a block and flushed (written out) when: the block is full, `fflush` is called, a new-line is written, or input is requested on a stream that is not fully buffered.

xviii. Whether a zero-length file can actually exist:

☞ Files of zero length can exist.

xix. The rules for composing valid filenames:

☞ Filenames may be up to 255 characters in length and may be composed of any printable characters other than backslash (\). If a path name is included in the file name, each component of the path name follows the same rules, and the components are separated by the '\' character.

xx. Whether the same file can be open multiple times:

☞ A given file may be open multiple times. If it is open multiple times for writing, it should be opened in append mode in all cases.

xxi. The effect of the `remove` function on an open file:

☞ An open file may be `removed`, but it is still accessible until it is no longer open.

xxii. The effect if a file with the new name exists prior to a call to the `rename` function:

☞ If a file already exists with the new name prior to a call to the `rename` function, the function returns failure, and sets errno to EEXIST.

xxiii. The output for `%p` conversion in the `fprintf` function:

☞ The %p conversion for `fprintf` and `fscanf` is equivalent to the `%lx` conversion.

xxiv. The input for %p conversion in the `fscanf` function:

☞ The %p conversion for `fprintf` and `fscanf` is equivalent to the `%lx` conversion.

xxv. The interpretation of a hyphen (-) character that is neither the first nor the last character in the scanlist for %[conversion in the `fscanf` function:

☞ In the `fscanf` function, a hyphen (-) character that is neither the first nor the last character for the %[conversion indicates an inclusive character range. That is, `%[-a-d]` and `%[-abcd]` are equivalent, as are `%[a-d-]` and `%[abcd-]`. Exception: The leading '^' character, indicating excluded characters, is never taken as part of a character range. Thus, `%[^-ab]` means all characters except '-', 'a', and 'b'.

xxvi. The value to which the macro `errno` is set by the `fgetpos` or `ftell` function on failure:

☞ `errno` is set to the value EIO on errors in `ftell` and `fgetpos`.

xxvii. The messages generated by the `perror` function:

☞ The various values of `errno` cause `perror` to print the following messages, or the message "no error" if `errno` is zero.

VALUE	MESSAGE
EACCES	"Permission denied"
EBADF	"Operation not possible on this file"
EBOUNDS	"Bounds check (index or subrange error)"
EDIVO	"Divide by Zero"
EDOM	"Input value not in function domain"
EINVAL	"Invalid argument"
EIO	"I/O error"
EMFILE	"Too many open files"
EOVFLO	"Overflow"
ERANGE	"Return value not in function range"

Other various values of `errno` cause `perror` to print the message contained in the system vector of message strings `sys_errlist` (see sections perror(3) and intro(2) in your UNIX documentation).

xxviii. The behavior of the `calloc`, `malloc`, or `realloc` function if the size requested is zero:

☞ Functions `calloc` and `malloc` return a null pointer if the size requested is zero and take no further action. Function `realloc` always frees the returned space, but returns a null pointer if the size requested is zero.

xxix. The behavior of the `abort` function with regard to open and temporary files:

☞ The `abort` function leaves buffered files unflushed but closes all files; temporary files are not deleted.

xxx. The status returned by the `exit` function if the value of the argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE`:

☞ The status sent to the operating system by `exit` is the parameter of the argument.

xxxi. The set of environment names and the method for altering the environment list used by the `getenv` function:

☞ The set of environment names provided to a program are the same as those in the environment of the process that spawns the program. (There are not necessarily any such names.) The environment list may be altered by declaring the variable `extern char **environ` and assigning the address of a new null-terminated array of string pointers to `environ`. Alternatively, individual string pointers within the original array may be replaced. Each string in the environment array must have the format given earlier. Neither the original array at `environ` nor any of its original strings may be freed by way of the function `free`.

xxxii. The contents and mode of execution of the string by the `system` function:

☞ The `system` command takes a string suitable for presentation to the Bourne shell as a command. A copy of the shell (/bin/sh) is spawned, and the string is given to it to execute (by way of the shell's -c flag).

xxxiii. The contents of the error message strings returned by the `strerror` function:

☞ The contents of the error messages returned by the `strerror` function are the same as for the `perror` function described previously.

xxxiv. The local time zone and Daylight Saving Time:

☞ The host operating system is queried for the local time zone and daylight savings time status, and the returned result is used.

xxxv. The era for the `clock` function:

☞ The host operating system is queried for the execution time used by the program, and the function `clock` uses that value. The era normally begins at program startup as defined by the host operating system.

Locale-Specific Behavior

This section defines the locale-specific behaviors of LPI-C ANSI library, as listed in section A.6.4 of the ANSI C standard.

- The only defined locale is the "C" locale.
- There are no assumed extensions to the execution character set.
- Printing is from left to right.
- The decimal point character is '.' (dot).
- The collation sequence of the execution character set is the ASCII collation sequence.
- For function `strftime`, the time and date formats are as given follows.
  - Abbreviated weekday names: "Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun".
  - Full weekday names: "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday".
  - Abbreviated month names: "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec".

- Full month names: "January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December".
- Format for "%c" conversion: Same as for function `asctime`.
- Format for "%p" conversion: "AM", "PM".
- Format for "%x" conversion: "dd Mon yyyy"
- Format for "%X" conversion: "hh:mm:ss":
- Format for "%Z" conversion, given hours west of Greenwich, half-hour increments:

```
0 :  GMT",      "GMT-0:30",   "GMT-1",    "GMT-1:30",
2 :  "GMT-2",   "GMT-2:30",   "GMT-3",    "NST",
4 :  "AST",     "GMT-4:30",   "EST",      "GMT-5:30",
6 :  "CST",     "GMT-6:30",   "MST",      "GMT-7:30",
8 :  "PST",     "GMT-8:30",   "YST",      "GMT-9:30",
10 : "HST",     "GMT-10:30",  "GMT-11",   "GMT-11:30",
12 : "GMT-12",  "GMT-12:30",  "GMT-13",   "GMT-13:30",
14 : "GMT-14",  "GMT-14:30",  "GMT-15",   "GMT-15:30",
16 : "GMT-16",  "GMT-16:30",  "GMT-17",   "GMT-17:30",
18 : "GMT-18",  "GMT-18:30",  "GMT-19",   "GMT-19:30",
20 : "GMT-20",  "GMT-20:30",  "GMT-21",   "GMT-21:30",
22 : "GMT-22",  "GMT-22:30",  "GMT-23",   "GMT-23:30"
```

Reference

ANSI Committee X3J11, *Draft Proposed American National Standard for Information Systems -- Programming Language C*, Document No X3J11/88-159, Dec 7, 1988.

Appendix D: LPI-C Compatibility Issues

This appendix describes LPI-C's extensions to support old-style C constructs.

LPI-C Extensions

In order for LPI-C to be able to correctly compile and execute the vast amounts of C code that exist with minimal change, and to be compatible with older pre-ANSI C implementations (most notably, PCC based compilers), a number of compatibility modes are supported. Listed below are some of the key non-ANSI or undefined-ANSI extensions that are supported by LPI's conforming ANSI C implementation. These extensions are provided by way of the compiler options for non-ANSI items, or as default behavior for undefined-ANSI items.

- Accept `long float` as a synonym for `double` in a declarator.
- Assign integer constants their types by old K&R rules [K&R Section 1 2.4.1] rather than ANSI rules.
- Perform the integral promotions according to "unsigned-preserving" rather than "value-preserving" rules.
- Accept the type `char *` as equivalent to the generic pointer type `void *`, without warning.
- Allow integral types other the `int` (signed or unsigned) to be declared as bit-fields, with appropriate maximum widths.
- Allow pernicious redefinitions of macros, with a warning.
- Allow innocuous redefinitions of function-like macros, which differ only in the spelling of of formal macro parameters. For example,

```
#define min(q,r)    (((q) <= (r)) ? (q)
: (r))
#define min(x,y)    (((x) <= (y)) ? (x)
: (y))
```

  would be silently accepted, without warning.

- Allow trailing text on a `#else` or `#endif` preprocessing directive, without warning [Standard Section 3.8.1].
- Expand function-like macro invocations with empty arguments as if the (empty) arguments consisted of no tokens. For example,

```
#define minus(x,y)    (x-y)
minus(,123)
```

  would yield "`(-123)`". ANSI C leaves the behavior of such a construct undefined.
- Perform formal macro parameter substitution within string literals [Rationale Section 3.8.3.2]. For example,

```
#define str(a)    "a: apple"
#define alpha     A
str(alpha)
```

  would yield "`"alpha: apple"`" rather than "`"a: apple"`", which is required by ANSI C. ANSI C provides a "stringize" operator (#) to accomplish this.
- Perform formal macro parameter substitution within character constants (exactly analogous to string literals).
- Use alternate scoping rules (that is, file scope) for functions and data objects declared as `extern` within an inner block scope.

Compatibility Options

For compatibility options with older pre-ANSI implementations (most notably PCC based compilers), refer to the "Compatibility Options" section in Chapter 1.

Old-Style C vs. ANSI C Integral Typing Rules and Promotions

The following tables summarize the integer constant typing rules for old-style C and ANSI C.

TABLE D-1 ANSI C Integral Constant Typing Rules

INTEGRAL CONSTANT	ASSIGNED TYPE (first in list in which they fit)
Unsuffixed decimal:	`int, long, unsigned long`
Unsuffixed octal/hex:	`int, unsigned int, long, unsigned long`
L suffixed:	`long, unsigned long`
U suffixed:	`unsigned, unsigned long`
L & U suffixed:	`unsigned long`

TABLE D-2 Old-Style C Integral Constant Typing Rules

INTEGRAL CONSTANT	ASSIGNED TYPE (first in list in which they fit)
Unsuffixed decimal:	`int, long`
Unsuffixed octal/hex:	`int, unsigned int, long`
L suffixed:	`long`

The following tables summarize the integral promotions for old-style C and ANSI C.

TABLE D-3 ANSI C Integral Promotions — Value-Preserving

OPERAND TYPE	OPERAND SIZE	PROMOTED TYPE
`char`	any	`int`
`short`	any	`int`
`int`	any	`int`
`unsigned char`	smaller than `int`	`int`
`unsigned char`	same as `int`	`unsigned int`
`unsigned short`	smaller than `int`	`int`
`unsigned short`	same as `int`	`unsigned int`
`unsigned int`	smaller than `int`	`int`
`unsigned int`	same as `int`	`unsigned int`

TABLE D-4 Old-Style C Integral Promotions — Unsigned-Preserving

OPERAND TYPE	OPERAND SIZE	PROMOTED TYPE
`char`	any	`int`
`short`	any	`int`
`int`	any	`int`
`unsigned char`	any	`unsigned int`
`unsigned short`	any	`unsigned int`
`unsigned int`	any	`unsigned int`

Index

LPI-C*

Language Reference Manual

This manual describes the implementation of LPI-C Version 1.

Contents

Chapter 3: Lexical Elements (Cont.)

Chapter 4: Preprocessing Directives

Chapter 5: Types

Chapter 5: Types (Cont.)

Chapter 6: Scope

Chapter 6: Scope (Cont.)

Chapter 7: Declarations

Chapter 7: Declarations (Cont.)

Chapter 8: Conversions

Chapter 9: Expressions

Chapter 9: Expressions (Cont.)

Chapter 10: Constant Expressions

Chapter 11: Statements

Chapter 12: ANSI C Compatibility Issues

Glossary

Appendices

Index

Preface: Using This Manual

Product Description

LPI-C is a fully conforming implementation of the C language as defined by the ANSI Committee X3J11 in *Draft Proposed American National Standard for Information Systems - Programming Language C, (Document No X3.159-1989, Dec 7, 1988)*. It passes the Plum Hall ANSI C Validation Test Suite.

The *LPI-C Language Reference Manual* describes the ANSI C language as implemented by Language Processors, Inc.

All LPI-C features adhere to this Standard. Implementation-defined behavior is explicitly noted; in cases where this behavior is machine-dependent, the user is referred to the *LPI-C User's Guide* for more information.

In addition, LPI-C contains several extensions for compatibility with earlier implementations of C, which are discussed in Chapter 12. There are several "compatibility modes" which may be selected at compile time to accomplish this. See your *LPI-C User's Guide* for additional information.

Related Documentation

For additional information on LPI-C, refer to the following manuals in the LPI-C documentation set:

- The *LPI-C User's Guide* describes how to compile, link, and run LPI-C programs. It also provides specific implementation information for use with your particular machine.

Intended Audience

This manual is written for experienced C programmers. It is not a tutorial, nor is it recommended for beginners.

Organization of Information

This manual consists of the following:

Chapter 1 describes the LPI-C translation environment.

Chapter 2 describes the LPI-C execution environment.

Chapter 3 describes lexical elements such as identifiers and constants.

Chapter 4 describes tokens and preprocessing directives.

Chapter 5 describes types and type derivations.

Chapter 6 describes the different scopes of identifiers, as well as function definitions.

Chapter 7 describes declarations and type specifiers.

Chapter 8 describes conversions and operands.

Chapter 9 describes expressions, operators, and function prototypes.

Chapter 10 describes constant expressions.

Chapter 11 describes statements.

Chapter 12 describes differences between ANSI C and earlier implementations of Kernighan and Ritchie C.

A glossary is provided for LPI-C specific terms.

Appendix A summarizes the language syntax.

Appendix B provides a list of identifiers, functions, and macros.

Appendix C describes implementation limits.

Appendix D describes compilation and numerical limits.

Appendix E describes unspecified and undefined behavior.

The index provides a quick reference for finding important LPI-C terms.

Syntax Conventions

The following syntax conventions are used in this manual.

1. Typewriter font is used for keywords, reserved words, and instances in which identifiers from programming examples are referred to in the text.

```
#include <stdio.h>
main ()
{
printf("This is typewriter font")
}
```

2. User-supplied information is indicated by italic type. For example:

constant-expression:
    conditional-expression

3. Alternative definitions are listed on separate lines, except when prefaced by the words "one of."

inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression | *exclusive-OR-expression*

4. An optional symbol is indicated by the subscript "*opt*," so that {!*expression sub opt*!} indicates an optional expression enclosed in braces.

Chapter 1: Translation Environment

Chapter 1: Translation Environment

Overview

This chapter describes the units and phases of the translation environment.

Translation Units

A translation unit consists of a source file, which contains the program text, along with any headers and source files that may be included by way of the preprocessing directive `#include`. The compilation units (that is, the resulting source text to be compiled) do not include any lines that were excluded by way of conditional inclusion preprocessing directives (for example, `#if`; see Chapter 4 for further discussion of preprocessing directives).

Translation units may be compiled separately, then saved individually or in libraries and later linked to produce an executable program. Previously compiled translation units are also referred to as object modules.

Separate translation units of a program can communicate at execution time through the following means:

- calls to functions whose identifiers have external linkage
- manipulation of objects whose identifiers have external linkage
- manipulation of data files

Translation Phases

The translation of a source file into an executable program is conceptually divided into a sequence of eight phases that are described in this section. Each phase performs, in order, a portion of the translation.

This is a conceptual model only; in actual practice, many of the phases are combined. For example, in many implementations, including LPI-C,

phases 1 through 3 are grouped together into what is referred to as scanning or lexical analysis; Phase 4 represents preprocessing; phases 5 through 7 are grouped into parsing or semantic analysis; and Phase 8 represents linking.

Note that this section makes reference to material that is fully described in later chapters of the book, especially the chapters covering lexical elements and preprocessing.

1. End-of-line Recognition and Trigraph Mapping

A new-line character is placed at the end of each physical line in the source file. On most UNIX-based systems, this is not relevant since the end-of-line is already represented by a single (ASCII) line-feed character. On most MS-DOS-based systems, this placement of the new line character simply involves replacing each (ASCII) carriage-return/line-feed pair with one new-line character.

Next, each trigraph sequence in the source file is converted to its corresponding single character equivalent. (Trigraphs are three-character sequences that begin with two question marks (??) and are discussed fully in Chapter 3.)

2. Line-splicing

Each occurrence of a backslash (\) character that is immediately followed by a new-line character is deleted, thereby splicing two physical source lines into one logical source line. (In diagnostic messages, LPI-C will always refer to the physical rather than the logical source line number.)

A non-empty source file must end in a new-line character, which itself is not immediately preceded by a backslash character.

Line-splicing is used primarily (but not exclusively) to write preprocessing macro definitions that cannot easily fit onto one physical line.

3. Decomposition into Preprocessing Tokens

The source file is broken down into a sequence of preprocessing tokens, new-lines, and other sequences of white-space, including comments.

Preprocessing tokens may consist of identifiers, string literals, character constants, preprocessing numbers, operators, punctuators, and header names, and are discussed fully in Chapter 4.

Other sequences of white-space may consist of spaces; horizontal tabs; form-feeds or vertical tabs (but not within preprocessing directives); or comments, each of which is replaced by one space character. Note that comments are not recognized within string literals, character constants, or comments (that is, comments do not nest).

LPI-C will convert sequences of white-space characters to one space character; other implementations may choose to retain the exact spelling of the white-space character sequence. (This distinction in conversions can only make a difference in the case of the macro expansion of a function-like macro invocation that uses the stringize (#) preprocessing operator.)

This decomposition of the source file into preprocessing tokens is context-dependent. For example, the recognition of a header name preprocessing token occurs only within a `#include` preprocessing directive. (See Chapter 4 for further discussion of preprocessing tokens and Chapter 3 for further discussion of white space.)

A source file may not end in a partial preprocessing token and may not end within a comment.

4. Preprocessing

Each preprocessing directive is processed and each macro invocation is fully expanded. Text inside conditionally excluded groups (that is, groups that are excluded by way of `#if`, `#ifdef`, or `#ifndef` directives) is ignored. A `#include` preprocessing directive will cause the named source file to be processed from Phase 1 through Phase 4 recursively, which, in effect, causes the named source file to be included as a part of the translation unit at the point at which the `#include` directive appeared. (See Chapter 4 for a complete discussion of preprocessing.)

5. **Escape Sequence Mapping**

   Each escape sequence within each string literal and character constant is converted to its corresponding single character in the execution character set. (See Chapter 3 for more on escape sequences.)

6. **String Literal Concatenation**

   Each sequence of adjacent string literals is concatenated to form a single string literal token. Terminating null characters from each adjacent string are not included, except at the end of the resulting string.)

7. **Syntactic and Semantic Analysis**

   At this point in the translation, white-space characters, which served to separate preprocessing tokens in phases 1 through 4, are no longer significant and are discarded.

   Preprocessing tokens are converted to tokens. In this conversion, keyword identifiers are recognized and preprocessing numbers are converted to integer or floating point constants. If any preprocessing tokens remain (for example, preprocessing numbers that are not converted to integer or floating point constants), an error will result and LPI-C produces an appropriate diagnostic. What remains after the conversion is simply a sequence of tokens, each of which has the lexical form of a keyword, an identifier, a string literal, a character constant, an integer constant, a floating point constant, an operator, or a punctuator.

   Syntactic and semantic analysis will be performed upon the sequence of tokens; if no fatal errors are encountered, compilation will be done and an object module will be produced.

8. **Linking**

   Any number of object modules may be linked (that is, combined) to form one final executable program. The linker program, usually provided by the system (for example, the `ld` program on UNIX systems), is responsible for resolving all external data object and/or function references and linking in appropriate library modules. To simplify the invocation of the system linker program, LPI-C provides a link script appropriate for each target system. (See your *LPI-C User's Guide* for further information.)

Translation Phases Examples

The specification of translation phases may be useful in resolving complex questions that can arise when coding in C. The following list addresses some of those questions, using the rules specified in the translation phases as described in the preceding section.

- A preprocessing directive (for example, a `#define` line) may be continued on the next physical line by preceding the end-of-line with a backslash, since Phase 2 effectively deletes occurrences of backslash/new-line, thus splicing physical source lines into one logical line before the preprocessing directive is recognized in Phase 4.
- A string literal token may also be continued onto the next physical line by preceding the end-of-line with a backslash, again since line-splicing is done in Phase 2 before token recognition in Phase 3. However, using the adjacent string literal concatenation feature is usually the preferred way of writing long string literals.
- Trigraph sequences are recognized and translated within string literals and character constants, since trigraph mapping in Phase 1 is done before token recognition in Phase 3.
- Since comments are replaced by one space character in Phase 3 before preprocessing directives are preprocessed in Phase 4, a preprocessing directive (for example, a `#define` line) may be continued onto the next physical line by using comments as in the following example:

```
#define MAX /* continued to the next line
                ... */ HEADROOM
```

- Again, since comments are replaced by one space character in Phase 3, the following construct:

```
a/* comment */b
```

  produces the two tokens, `a` and `b`, rather than the one token `ab`.

- A comment is not recognized as a comment (that is, it is not skipped) when it is within a string literal or character constant, since string literals and character constants are tokens themselves and their contents are not examined for comments.

  For example, the following string literal token:

  ```
  "a/* comment */b"
  ```

  would remain the same throughout translation and would not be translated as `"a b"`.

  This is actually a result of the definition of a comment given in Chapter 3, and not a result of the rules of translation phases, since comment and token recognition both occur in Phase 3.

- Since token decomposition is performed in Phase 3 before processing occurs in Phase 4, the following construct:

  ```
  #define APLUS +a
  +APLUS
  ```

  yields the following three tokens: `+`, `+`, and `a`, rather than yielding two tokens: `++` and `a`.

- Since tokens are recognized in Phase 3 before macro expansion in Phase 4, defined macro names are not expanded within string literals or character constants. For example:

  ```
  #define MYSTRING string
  printf ("MYSTRING")
  ```

  would not print `string`, but rather `MYSTRING`.

- Macros are expanded in `#include` lines. For example:

  ```
  #define MY_DEF_HEADER  "mydefs.h"
  #include MY_DEF_HEADER
  ```

is equivalent to:

```
#include "mydefs.h"
```

- Adjacent strings are not concatenated in `#include` directives, since adjacent string literal concatenation occurs in Phase 5, whereas `#include` directives are processed earlier in Phase 4. For example, the following construct is illegal:

```
#define MY_DEF_HEADER  "defs"
#define EXTENSION  ".h"
#include MY_DEF_HEADER EXTENSION
```

- During preprocessing in Phase 4, there is no notion of keywords since keywords are not recognized until Phase 7. Therefore, keywords may be redefined just as any other identifier. For example, the following construct is legal:

```
#define int unsigned int
```

- Macros may not be defined to represent opening and closing comment delimiters, since all comments are mapped to one space character in Phase 3, which occurs before preprocessing (and `#define` processing) in Phase 4. The characters `/*` will serve to open a comment and the characters `*/` will serve to close a comment. For example,

```
#define START_COMMENT /*
#define END_COMMENT   */
```

is equivalent to the following:

```
#define START_COMMENT
```

Chapter 2: Execution Environment

Chapter 2: Execution Environment

Overview

This chapter describes the execution environment: program startup, execution, and termination.

Program Startup

All objects with initial values that are in static storage are initialized before program startup. The function named **main** is called at program startup. There is no function prototype for **main**. Therefore, parameters are not necessary when defining the **main** function, as in the following examples:

```
int main(void) {     }
```

or

```
main ( )     {    }
```

Alternatively, two parameters may be used when defining **main**, as in the following example, which uses the parameters named **argc** and **argv**.

```
int main(int argc, char *argv[]) {      }
```

Parameters are local to the function in which they are declared; therefore, any names can be given to the parameters.

The defined parameters to the **main** function in the preceding example (**argc** and **argv**) must adhere to the following constraints:

- **argc** will have a nonnegative value representing the size of the **argv** array.
- If the value of **argc** is zero, then the value of the **argv** is undefined and should not be referenced.

- If `argc` has a value that is greater than zero, the array members `argv[0]` through `argv[argc-1]` will contain pointers to null-terminated strings. The implementation provides the program with information that has been determined before program startup by predefining values for those strings. Command-line arguments before program startup allow the program to access information that was determined from elsewhere in the host environment.
- `argv[argc]` will be a null pointer.
- If the value of `argc` is greater than zero, the program name is represented by the string pointed to by `argv[0]`.
- If the value of `argc` is greater than one, the strings pointed to by `argv[1]` through `argv[argc-1]` represent the program parameters.
- The program can modify the parameters `argc` and `argv`, as well as the strings pointed to by the `argv` array during program execution, and rely on the most recent modifications that were properly saved.

For example, on a typical UNIX system, your program called `progname` can be invoked with the strings `argone` and `argtwo` with the command:

```
progname argone argtwo
```

When the program is invoked, the array of pointers will appear as in the following figure:

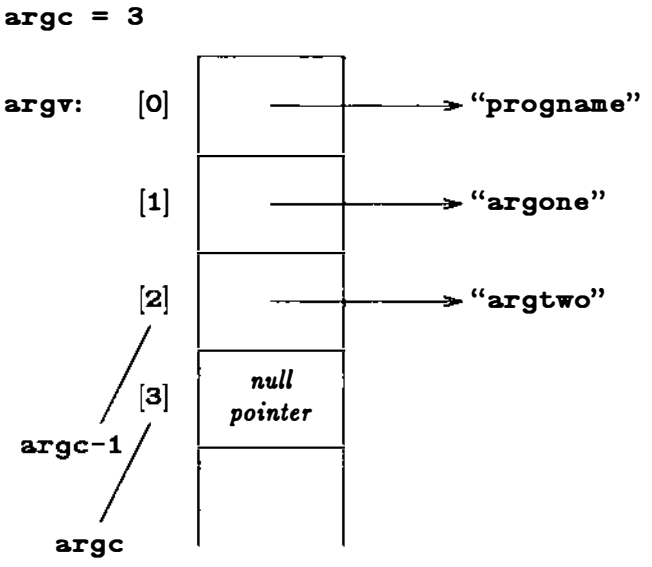


FIGURE 2-1 Array of Pointers

Program Execution

To complete execution, a program may use any of the functions, macros, type definitions, and objects described in the *LPI-C Library Reference Manual.*

Implementation Semantics

ANSI C allows the implementation to define a one-to-one correspondence between abstract and actual semantics. That is, at each sequence point, the values of the actual objects would agree with those values specified by the abstract semantics. (See Chapter 9 for further discussion of sequence points.)

If the abstract and actual semantics agree, then the keyword `volatile` would be unnecessary. (See Chapter 7 for further discussion of `volatile`.) ANSI C also allows an implementation to perform optimizations such that the actual semantics agree with the abstract semantics only when function calls are made across translation unit

boundaries. If the called functions and calling functions are in different translation units, the values of the following would agree with the abstract semantics at the time of function entry or function return:

- all externally linked objects
- all objects accessible by way of pointers

At the time of function entry, the values of the following would agree with the abstract semantics:

- the values of the parameters of the called function
- all objects accessible by way of pointers

Note that explicit specification of `volatile` storage would be required when objects that are referred to by interrupt service routines are activated by the signal function.

Signals and Interrupts

A function can be interrupted at any time by a signal and can be called by a signal handler. If either a signal or a signal handler (or both) interrupts a function, there will be no modification in whatever processing had already been completed prior to the interruption, such as active function invocations and associated automatic data.

Static data may be modified by such interruptions, as library functions may not necessarily be reentrant.

Program Termination

A program is terminated in one of two ways:

- when the standard library `exit` function is called (see the *LPI-C Library Reference Manual*)
- when the `main` function returns (that is, upon the initial call to the `main` function)

Chapter 3: Lexical Elements

Chapter 3: Lexical Elements

Overview

This chapter describes the following lexical elements from which a program is constructed: characters, tokens, keywords, identifiers, constants, string literals, operators, and punctuators.

Source and Execution Character Sets

There are two sets of program characters in LPI-C:

- the set in which source files are written
- the set interpreted in the execution environment

In most implementations of LPI-C, both the source and the execution character sets are the ASCII character set. (See Appendix C, item 4.i, in the *LPI-C User's Guide.*)

Characters Included in Both Sets

Each of these character sets has the following members:

- the 52 upper- and lowercase letters of the English alphabet:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

- the 10 decimal digits:

```
0 1 2 3 4 5 6 7 8 9
```

- the 29 graphic characters:

!	exclamation point	"	double quote
#	number sign	%	percent
&	ampersand	'	single quote
/	slash	\	backslash
(	left parenthesis	)	right parenthesis
*	asterisk	+	plus
,	comma	-	hyphen or minus
.	period	:	colon
;	semicolon	<	less than
=	equal	>	greater than
?	question mark	[	left bracket
]	right bracket	^	caret
_	underscore	~	tilde
|	vertical bar	{	left curly brace
		}	right curly brace

In addition, both the source and execution character sets contain the following:

- The space character.
- The control characters representing the horizontal tab, the vertical tab, and form feed.
- The end of line indicator, which will be treated as a single new-line character. (For example, on most UNIX systems, the end of line indicator is represented by the ASCII value 0xA, or line-feed. However, on many MS-DOS systems, the end-of-line is indicated by the sequence 0x0D0A, or carriage return/line feed.)

Additional Characters in the Execution Set

In addition to the preceding list of characters, the execution character set contains the following control characters: alert, backspace, carriage return, and new line (see the "Escape Sequences" section later in this chapter).

When a member of the execution character set appears in a character constant or string literal, it can be represented by its corresponding member of the source character set, or it can be represented by one of the escape sequences that contains the backslash (\), followed by one or more characters. If characters that are not in the execution character set are found in a source file, the behavior is undefined.

The null character terminates a character string literal. The null character has all bits set to 0.

Trigraph Sequences

When any of the following sets of three characters, called trigraph sequences, appears in the source file, that set is then replaced with a single character. This allows the user to input characters that are not defined in the ISO 646-1983 Invariant Code Set, which is a subset of the seven-bit ASCII code set.

TRIGRAPH SEQUENCE	SINGLE-CHARACTER REPLACEMENT
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

If a program contains a question mark (?) that does not begin one of the trigraphs in the preceding table, that question mark is not affected.

For example, the following source line

```
printf("This is not two question marks:???/n");
```

becomes (after replacement of the trigraph sequence ??/)

```
printf("This is not two question marks:?\n");
```

Comments

A comment is introduced by the characters `/*`, except within a character constant, a string literal, or a comment. During processing, a comment is examined only to identify any multibyte characters and to find the characters `*/` at the end of the comment.

White Space

White space is generally used to separate preprocessing tokens so that individual preprocessing tokens will not be parsed as one token. As an exception, white space may only be included as part of a header name or as part of a string literal or a character constant.

White space consists of any of the following:

- one or more of the following characters: space, horizontal tab, new-line, vertical tab, and form-feed
- a comment (each comment is replaced by one white space character in translation phase 3)
- a combination of comments and white-space characters

Note that all white space characters can be regarded as the same except in the preprocessing phase, where new-lines are significant, and form-feed and vertical tabs are not allowed in preprocessor directives.

White Space Examples

To avoid unexpected results that may be produced by parsing individual preprocessing number tokens as if they were one complete token, it is recommended that white space be used liberally. When, for example, the program fragment `8fx` is parsed, it is parsed as one single preprocessing number token, which is not a valid floating or integer constant token and will result in an error in translation phase 7.

If, on the other hand, the pair of preprocessing tokens `8` and `fx` are parsed, a valid expression might be produced (for example, if `fx` were a macro defined as `+1`).

When the program fragment `x+++++y` is parsed, it is parsed as `x ++ ++ + y`, (with white space before the final `+`) which violates a constraint on increment operators. The parse `x ++ + ++ y` (with white space before the final two `++`), however, might yield a correct expression.

Unexpected results can be produced by lack of white space in certain preprocessing number token constructs. For example, the program fragment `0x123E+abcde` is not parsed as the three tokens `0x123E` `+` `abcde` as might be expected. Instead, it is parsed as one preprocessing number token, which is not a valid token and will result in an error in phase 7. Placing a white space in front of the `+` will yield the expected result.

Tokens

A token is defined as the minimal lexical element of the language in translation phases 7 and 8. (Preprocessing tokens are discussed in Chapter 4. See Chapter 1 for discussion of translation phases.)

A token is one of the following:

- keyword
- identifier
- constant
- string literal
- operator
- punctuator

Token Syntax

token:
keyword
identifier
constant
string-literal
operator
punctuator

Keywords

The following is a list of words, which are used entirely in lowercase, that are reserved for use in LPI-C in translation phases 7 and 8 as described in Chapter 1.

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Identifiers

An identifier consists of a sequence of nondigit characters and digits, with the first character a nondigit character. (Nondigit characters include the underscore (_) as well as lowercase and uppercase letters.)

An identifier refers to one of the following:

- object
- function
- tag
- member of a structure, union, or enumeration
- `typedef` name
- label name
- macro name
- macro parameter
- enumeration constant

Identifier Names

Identifier names that differ in a significant character constitute different identifiers. LPI-C assumes identifier names that differ in an insignificant character to be the same identifier.

An identifier name must not have the same sequence of characters as a keyword; identifiers are also case-sensitive.

LPI-C defines the first 256 characters of an internal name (that is, a macro name or an identifier that does not have external linkage) to be significant, exceeding the minimum of 31 required by the ANSI C Standard.

ANSI C allows the implementation to restrict the length of an external name (an identifier that has external linkage) to six significant characters. Distinctions of uppercase and lowercase for external names may also be ignored. (See Appendix C, item 3.iii, in the *LPI-C User's Guide* for further information on LPI-C's implementation of this feature.)

Identifier Syntax

identifier:
nondigit
identifier nondigit
identifier digit

nondigit: one of

```
_  a  b  c  d  e  f  g  h  i  j  k  l  m
   n  o  p  q  r  s  t  u  v  w  x  y  z
   A  B  C  D  E  F  G  H  I  J  K  L  M
   N  O  P  Q  R  S  T  U  V  W  X  Y  Z
```

digit: one of

```
0  1  2  3  4  5  6  7  8  9
```

Constants

This section describes the forms and values of the following constants:

- floating constants
- integer constants
- enumeration constants
- character constants

Constant Syntax

constant:
floating-constant
integer-constant
enumeration-constant
character-constant

Floating Constants

A floating constant contains the following components:

- a significand in a floating constant, which is a decimal rational number
- an optional exponent part
- an optional suffix that specifies the constant type
- either the whole-number part or the fraction part of the constant
- either the period or the exponent part of the constant

The significand may consist of the following:

- a digit sequence representing the whole-number part
- a period (.) following the digit sequence
- a digit sequence representing the fraction part, following the period

The optional exponent part of the floating constant consists of the following:

- e or E
- an exponent that can consist of a signed digit sequence, which is a decimal integer, following the e or E

The following table describes optional integer suffixes that specify the constant type.

SUFFIX	INTEGER TYPE
unsuffixed floating constant	`double`
suffixed by the letter `f` or `F`	`float`
suffixed by the letter `l` or `L`	`long double`

Floating Constant Syntax

floating-constant:
  fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
  digit-sequence$_{opt}$. *digit-sequence*
  digit-sequence .

exponent-part:
  `e` *sign*$_{opt}$ *digit-sequence*
  `E` *sign*$_{opt}$ *digit-sequence*

sign: one of
  `+` `-`

digit-sequence:
  digit
  digit-sequence digit

floating-suffix: one of
  `f` `l` `F` `L`

Integer Constants

An integer constant has the following characteristics:

- begins with a digit
- has no period or exponent part (unlike a floating constant)
- may have a prefix to specify the base

- may have a suffix to specify the type of the integer constant

The following table describes the components of specific integer constants.

INTEGER CONSTANT	CHARACTERISTICS
decimal	Begins with a nonzero digit Consists of a sequence of decimal digits Is computed base 10
octal	Consists of the prefix 0 Prefix 0 may be followed by a sequence of the digits 0 through 7 Is computed base 8
hexadecimal	Consists of the prefix 0x or 0X Prefix may be followed by a sequence of the decimal digits and the letters a (or A) through f (or F) with values 10 through 15 Is computed base 16

Integer Constant Types

The following table shows the possible types that may be attributed to an integer constant.

The attributed type is the first type in which its value can be represented that appears in the corresponding list on the right.

SUFFIX	INTEGER TYPE
unsuffixed decimal	`int, long int, unsigned long int`
unsuffixed octal or hexadecimal	`int, unsigned int, long int, unsigned long int`

SUFFIX	INTEGER TYPE
suffixed by the letter u or U	`unsigned int, unsigned long int`
suffixed by the letter l or L	`long int, unsigned long int`
suffixed by both the letters u or U and l or L	`unsigned long int`

Integer Constant Syntax

integer-constant:
decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:
nonzero-digit
decimal-constant digit

octal-constant:
`0`
octal-constant octal-digit

hexadecimal-constant:
`0x` *hexadecimal-digit*
`0X` *hexadecimal-digit*
hexadecimal-constant hexadecimal-digit

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

octal-digit: one of
`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
    unsigned-suffix long–suffix$_{opt}$
    long-suffix unsigned–suffix$_{opt}$

unsigned-suffix: one of
    `u  U`

long-suffix: one of
    `l  L`

Character Constants

A character constant has the following characteristics:

- It has type `int`.
- It has a numerical value: the character is interpreted as an integer of the execution character set (for example, in ASCII, 'A' is interpreted as 65).
- It is a sequence of one or more multibyte characters enclosed in single-quotes, as in 'a' or 'xz'.
- The elements of a sequence of integer character constants are any members of the source character set, which are in turn mapped by the compiler in an implementation-defined manner to the members of the execution character set. (See Appendix C, item 4.v, in the *LPI-C User's Guide.*) There are a few exceptions with escape sequences detailed later in this chapter.
- If an integer character constant contains more than one character, then its corresponding value is implementation-defined. LPI-C, for example, will give the multibyte character `'ab'` a value of `0x3132`, assuming an ASCII execution character set.
- If an integer character constant contains a character (or an escape sequence) that is not represented in the basic execution character set, then its corresponding value is implementation-defined. LPI-C will ignore the \ for undefined escape sequences. For example, `'\z'` is translated to `'z'`.
- If an integer character constant contains a single character or escape sequence, its value is the same as it would be if an object with type

`char` were converted to type `int`, assuming that the object has the same value as the integer character constant's single character or escape sequence.

- A wide character constant has the same characteristics as an integer character constant, but has the letter L as a prefix.
- A wide character constant has the integral type `wchar_t`, as defined in the `<stddef.h>` header.
- If a wide character constant contains a single multibyte character that maps into a member of the extended execution character set, the value is the wide character (code) corresponding to that multibyte character, as defined by the `mbtowc` function. (See the *LPI-C Library Reference Manual* for more information on this function.) LPI-C defines the current locale as the "C" locale.
- If a wide character constant contains more than one multibyte character, or if it contains a multibyte character or escape sequence that is not represented in the extended execution character set, its value is implementation-defined. (See Appendix C, item 4.vii, in the *LPI-C User's Guide*.)

Escape Sequences

LPI-C supports the escape sequences shown in the following table:

TABLE 3-1 Escape Sequences

ESCAPE SEQUENCES	REPRESENTATION
\'	single-quote '
\" or "	double-quote "
\? or ?	question-mark ?
\\	backslash \
octal digits	octal integer
hexadecimal digits	hexadecimal integer

Octal and hexadecimal escape sequences are both terminated by the first non-octal or first non-hexadecimal digit, respectively, except that an octal escape sequence is comprised of at most three digits. (See the following section, "Octal and Hexadecimal Escape Sequences.")

There are also escape sequences that consist of the backslash \ followed by a lower-case letter: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`. (See the section "Nongraphic Display Characters" later in this chapter.)

Any other escape sequence will produce undefined behavior.

Octal and Hexadecimal Escape Sequences

The value of an octal or hexadecimal escape sequence must meet the following constraints:

- For an integer character constant, the value must be in the range of representable values for the type `unsigned char`.
- For a wide character constant, the value must be in the range of representable values for the unsigned type corresponding to `wchar_t`.

In an octal escape sequence, all octal digits following the backslash together specify the value of a single character for an integer character constant (or a single wide character for a wide character constant).

In a hexadecimal escape sequence, all hexadecimal digits following the backslash and the letter `x` together specify the value of a single character for an integer character constant (or a single wide character for a wide character constant).

Nongraphic Display Characters

Nongraphic characters in the execution character set are represented by a corresponding set of alphabetic escape sequences.

These escape sequences can be used, with `printf`, for example, to affect display devices as follows:

`\a`	(alert) Does not change the current display position. Sounds an audible alert or produces a visible alert.
`\b`	(backspace) Moves the current position to the previous position on the current line.
`\f`	(form feed) Moves the current display position to the first position at the beginning of the next logical page.

`\n`	(new line) Moves the current display position to the first position of the next line.
`\r`	(carriage return) Moves the current display position to the first position of the current line.
`\t`	(horizontal tab) Moves the current display position to the next horizontal tab position on the current line.
`\v`	(vertical tab) Moves the current display position to the first position of the next vertical tab position.

Character Constant Syntax

character-constant:
  ´ *c-char-sequence* ´
  L´ *c-char-sequence* ´

c-char-sequence:
  c-char
  c-char-sequence c-char

c-char:
  any member of the source character set except
    the single-quote ´, backslash \, or new-line character
  escape-sequence

escape-sequence:
  simple-escape-sequence
  octal-escape-sequence
  hexadecimal-escape-sequence

simple-escape-sequence: one of
  `\´  \"  \?  \\`
  `\a  \b  \f  \n  \r  \t  \v`

octal-escape-sequence:
  `\` *octal-digit*
  `\` *octal-digit octal-digit*
  `\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
  `\x` *hexadecimal-digit*
  hexadecimal-escape-sequence hexadecimal-digit

Character Constant Examples

The null character is generally represented by the construction `'\0'`.

LPI-C uses a two's-complement representation for integers and eight bits for objects that have type `char`. In LPI-C, since type `char` has the same range of values as `signed char`, the integer character constant `'\xFF'` has the value −1.

The integer character constant with the following hexadecimal construction:

```
'\x113'
```

will specify only one character instead of two, regardless of whether or not an object with type `char` is represented in eight bits.

If, however, the programmer intended to specify an integer character constant containing two characters with the following values:

```
0x11 and '3'
```

then, because only a non-hexadecimal character terminates a hexadecimal escape sequence, the following construction must be written:

```
'\0213'
```

which is equivalent to `'\021' '3'`, since octal escape sequences are comprised of at most three digits. Likewise, `L'\1254'` will result in a wide character constant having the values `'125' '4'` for the reasons described above.

Enumeration Constants

An enumeration constant in C has the same lexical structure as an identifier but has type `int`. (See Chapter 5 for further information.)

Enumeration Constant Syntax

enumeration-constant:
identifier

String Literals

A string literal is one of the following:

- a character string literal, which is a sequence of zero or more multibyte characters enclosed in double-quotes, as in `"xyz"`
- a wide string literal, which is the same, except prefixed by the letter L

For each element of the sequence in a character string literal or a wide string literal, the same constraints must be met as if it were in an integer character constant or a wide character constant. The only exceptions to this rule are that single-quote ' can be represented in two different ways, by itself or by the escape sequence `\'`, and that double-quote " can only be represented by the escape sequence `\"`.

If sequences of adjacent character string literal tokens (or adjacent wide string literal tokens) specify multibyte character sequences, then those multibyte character sequences are concatenated into a single multibyte character sequence in translation phase 6. (A character string literal adjacent to a wide string literal token will produce undefined behavior.)

The multibyte character sequences, as described previously, will have a `'\0'` byte appended in translation phase 7, which, in turn, will be used to initialize an array. That array is of `static` storage duration and is just long enough to contain the multibyte character sequence.

Individual bytes of the multibyte character sequence can be used to initialize the array elements for character string literals, which have type `char`. Note that string literals are also allowed to have embedded `'\0'` escape sequences.

A sequence of wide characters that corresponds to a multibyte character sequence is used to initialize the array elements of wide string literals, which have type `wchar_t`.

String Literal Syntax

string-literal:
"*s–char–sequence*$_{opt}$"
L"*s–char–sequence*$_{opt}$"

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

String Literal Example

Just prior to concatenation of adjacent string literals, escape sequences are converted into single members of the execution character set. For example, the following pair of adjacent character string literals,

```
"\x24" "7"
```

produces a single character string literal that consists of the two characters whose values are \x24 and '7'.

Operators

Each of the operators listed in the following syntax evaluates an operand to produce a value.

The following operators only occur in pairs, although each pair may be separated by other expressions.

```
[ ]

( )

? :
```

Only macro-defining preprocessing directives use the operators # and ##. (See Chapter 4 for further discussion of preprocessing directives.)

Operator Syntax

operator: one of

```
[  ]  (  )  .  ->
++  --  &  *  +  -  ~
!  sizeof  /  %
<<  >>  <  >  <=  >=  ==  !=
^  |  &&  ||  ?  :
=  *=  /=  %=  +=  -=
<<=  >>=  &=  ^=  |=
,  #  ##
```

Punctuators

Although a punctuator sometimes represents an operator, generally a punctuator does not evaluate an operand.

The following punctuators only occur in pairs, although they can be separated from each other by expressions, declarations and statements:

```
[ ]

( )

{ }
```

Only preprocessing directives use the punctuator `#`.

Punctuator Syntax

punctuator: one of

```
[  ]  (  )  {  }  *  ,  :  =  ;  ...  #
```

Header Names

Header name preprocessing tokens only appear within a `#include` preprocessing directive. (See Chapter 4 for further discussion of preprocessing directives.)

Chapter 4: Preprocessing Directives

Chapter 4: Preprocessing Directives

Overview

This chapter describes preprocessing tokens and preprocessing directives.

Preprocessing tokens are the minimal set of lexical elements, in translation phases 3 through 6, from which a program is constructed. (Tokens are the minimal lexical elements of the language in translation phases 7 and 8.)

Preprocessing directives are used to direct the compiler to include certain files or to substitute specific macros, and to include or exclude specific lines from source files.

Preprocessing Tokens

A preprocessing token is one of the following:

- header name
- identifier
- preprocessing number
- character constant
- string literal
- operator
- punctuator
- single non-white-space character that differs from the other preprocessing token categories in this list, with the exception of a single quote (') or a double quote (") character

Preprocessing Directives

ANSI C preprocessing directives and operators are listed in Table 4-1 and Table 4-2.

By using preprocessing directives, the compiler can be directed to include named files and substitute macros, as well as to include or exclude specific sections, of source files as needed.

Because preprocessing occurs during the first four phases of the translation phases described in Chapter 1, it can be thought of as a completely separate phase of translation. In fact, with LPI-C, it is possible to obtain the source output of the preprocessing phase (that is, with all macros expanded and files included). This procedure can be useful for debugging purposes. (The source output from the preprocessing phase can be subsequently compiled.)

A preprocessing directive consists of a sequence of preprocessing tokens. That sequence begins with a # preprocessing token, which is either the first character in the source file (and may also be preceded by white space), or which follows a white space containing at least one new-line character.

A preprocessing directive is ended by the first new-line character that appears after the introductory # preprocessing directive.

Form-feed and vertical tab white-space characters may not appear in a preprocessing directive. However, space and horizontal-tab white-space characters may appear within a preprocessing directive, and trailing white space is ignored. This means that preprocessing directives may be simply thought of as lines in which the first non-white space character is the # preprocessing token.

The following table summarizes the preprocessing directives and operators.

TABLE 4-1 Preprocessing Directives

DIRECTIVE	
`#define`	Defines an object-like macro or a function-like macro.
`#undef`	Undefines (that is, removes) a macro name definition.
`#include`	Includes the text of a specified source file or header file.
`#if`	Conditionally includes or excludes subsequent source text, based on whether or not a specified expression is true or false.
`#ifdef`	Conditionally includes or excludes subsequent source text, based on whether or not a specified name is defined as a macro.
`#ifndef`	Conditionally includes or excludes subsequent source text, based on whether or not a specified name is not defined as a macro.
`#elif`	Alternatively includes or excludes subsequent source text, based on whether or not a specified expression is true or false.
`#else`	Alternatively includes or excludes subsequent source text, based on whether or not a previous matching `#if`, `#ifdef`, `#ifndef`, or `#elif` directive, is true or false.
`#endif`	Terminates a conditionally included or excluded group that began with a matching preceding `#if`, `#ifdef`, or `#ifndef` directive.
`#line`	Forces the translator to behave as if the current line number and, optionally, the source file name, are those specified in the directive (for diagnostic purposes).
`#error`	Causes the translator to produce an error message containing the specified text.

TABLE 4-1 Preprocessing Directives (Cont.)

DIRECTIVE	
`#pragma`	Causes the translation to perform some implementation-defined task.

TABLE 4-2 Preprocessing Operators

OPERATOR	
`defined`	Evaluates to true or false, based on whether or not a specified name is defined as a macro (for use within a `#if` or `#elif` directive).
`##`	Concatenates tokens in a macro definition (token-paste).
`#`	Creates a string-literal consisting of the tokens in a macro function argument (stringize).

Preprocessing Syntax

preprocessing-token:
header-name
identifier
pp-number
character-constant
string-literal
operator
punctuator
each non-white-space character that cannot be one of the above

preprocessing-file:
$group_{opt}$

group:
group-part
group group-part

group-part:

pp-tokens$_{opt}$ *new-line*
if-section
control-line

if-section:

if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:

 # if *constant-expression new-line group*$_{opt}$
 # ifdef *identifier new-line group*$_{opt}$
 # ifndef *identifier new-line group*$_{opt}$

elif-groups:

elif-group
elif-groups elif-group

elif-group:

 # elif *constant-expression new-line group*$_{opt}$

else-group:

 # else *new-line group*$_{opt}$

endif-line:

 # endif *new-line*

control-line:

 # include *pp-tokens new-line*
 # define *identifier replacement-list new-line*
 # define *identifier lparen identifier-list*$_{opt}$**)**
replacement-list new-line
 # undef *identifier new-line*
 # line *pp-tokens new-line*
 # error *pp-tokens*$_{opt}$ *new-line*
 # pragma *pp-tokens*$_{opt}$ *new-line*
*new-line*

lparen:

the left-parenthesis character without preceding white-space

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

Source File Inclusion

One of the most common preprocessing directives is the `#include` directive. This directive specifies that the contents of a designated source file (or standard header) should be included in the source file at the point where the directive appeared. Typically, the designated source file to be included contains declarations, `typedef`s, and macro definitions to be used by a number of source files in a program, thereby reducing duplication of code and increasing program modularity.

The following three forms of the `#include` preprocessing directives may be used:

1. The following `#include` preprocessing directive form will search for a header that is uniquely identified by the name appearing between the `<` and `>` delimiters:

   `# include <`*h-char-sequence*`>` *new-line*

   For example:

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

   Once found, the entire contents of the header will be substituted for the directive itself.

   In general, LPI-C will search in the following sequence of places:

   - First, if the source file name specified is fully qualified by a directory path name (for example, if it begins with a slash (/) on UNIX systems), then the file will be searched for in the specified directory only.

- Otherwise, if not found in the specified directory, the file will be searched for within the directories specified on the command line by way of the `-ipath` option.
- Otherwise, if not found, then the file will be searched for within the standard LPI-C include file directory (which contains the ANSI C standard header files), or, if the `-syspath` option was given on the command line, within the standard system include file directory (for example, /usr/include on most UNIX systems).
- Otherwise, the search has failed and a diagnostic message will be issued.

2. A `#include` preprocessing directive of the following form will be replaced by the contents of the source file that is identified by the name between the " and " delimiters.

   `# include "`*q-char-sequence*`"` *new-line*

   For example:

   ```
   #include "data.h"
   ```

   Once found, the entire contents of the source file will be substituted for the directive itself.

   In general, LPI-C will search in the following sequence of places:

   - First, if the source file name specified is fully qualified by a directory path name (for example, if it begins with a slash (/) on UNIX systems). Then the file will be searched for in the specified directory only.
   - Otherwise, the file will be searched for in the same directory in which the including file resides.
   - Then, if not found, it will be searched for as if it had been included by way of the first method.

3. In the following form of the `#include` preprocessing directive, the preprocessing tokens after `include` in the directive are processed just as in normal text. That is, macros are fully

expanded. After macro expansion, the resulting directive must match one of the two previously described `#include` directive forms.

```
# include pp-tokens new-line
```

For example:

```
# if defined (M68_TARGET)
# define TARGET_IN "M68.in"
# elif defined (I386_TARGET)
# define TARGET_IN "I386.in"
# elif defined (M88_TARGET)
# define TARGET_IN "M88.in"
# elif defined (SPARC_TARGET)
# define TARGET_IN "sparc.in"
# endif

# include TARGET_IN
```

For further information concerning methods of source file search, see Appendix C, item 13.ii, in the *LPI-C User's Guide* for full details.

Header Names

Header name preprocessing tokens are recognized only within a `#include` preprocessing directive. This section describes the structure and syntax of header names. (The interpretation of header names is discussed in the previous section.)

The two forms of header names are shown in the following syntax.

Header Syntax

header-name:
  <*h-char-sequence*>
  "*q-char-sequence*"

h-char-sequence:
  h-char
  h-char-sequence h-char

h-char:
any member of the source character set except the new-line character and >

q-char-sequence:
q-char
q-char-sequence q-char

q-char:
any member of the source character set except the new-line character and "

The manner in which the sequences in both forms of header names are mapped to headers or external source files is implementation-defined, as is the way in which the preprocessing tokens between a < and a > preprocessing token, or two " characters, are combined into a header name preprocessing token. (See Appendix C, item 13.iv, in the *LPI-C User's Guide.*)

LPI-C does the following:

- A source filename specified within double quotes is taken exactly as it is written. For example, the following expression:

```
#include " f/* hi*/ ile\'x'.h"
```

  specifies the following file name:

```
`` f/*  hi*/ ile\'x'.h''
```

- A source header name specified between a < and > is created by concatenation of the individual preprocessing tokens that appear between the < and >.

  For example, the following expression:

```
#include < f/* hi*/ ile\'x'.h>
```

  specifies the header with the following name:

```
`` file.h ''
```

The behavior is undefined in the following cases:

- if the characters ' \ " or /* occur in the sequence between the < and > delimiters
- if the characters ' \ or /* occur in the sequence between the " delimiters
- if the characters correspond to escape sequences (since \ causes undefined behavior)

Macro Definitions

This section describes how to define object-like and function-like macros, as well as the method by which they are invoked and expanded.

A name may be defined to be an object-like or function-like macro through the use of the `#define` preprocessing directive. A macro name may be redefined as long as the following holds true:

- The second definition is the same kind of macro as the first (that is, either object-like or function-like).
- The definition of the two replacement lists in each macro definition are identical. Two replacement lists are identical if they consist of exactly the same sequence of preprocessing tokens and white space separation (where all non-empty sequences of white space within a replacement-list are considered identical).
- The number and names of macro parameters (for function-like macros) are the same.

Object-Like Macros

An object-like macro is defined using the following form of preprocessing directive:

`#define` *identifier replacement-list new-line*

Once such a definition is made, subsequent occurrences (that is, invocations) of the named identifier will be replaced by the sequence of preprocessing tokens that constitute the replacement-list (which may consist of no tokens), after any token-pasting takes place, as described below. This process is called macro expansion.

One of the more common uses of an object-like macro is to define manifest constants, and thereby avoid the generally poor practice of using hard coded constants throughout the source code, as illustrated in the following example:

```
#define MAX_HASH_TABLE_SIZE 1031

char *HashTable [MAX_HASH_TABLE_SIZE];
```

Note that occurrences of a named identifier within a string-literal will not be substituted since the string-literal is itself a token and its contents are not examined for macro expansion. The same is true for character constants.

Function-Like Macros

A function-like macro is defined using the following form of preprocessing directive:

#define *identifier*(*identifier-list*) *replacement-list new-line*

Note
No white space may occur between the identifier and the left parenthesis token.

The identifier list specifies zero or more comma-separated macro parameters that may be used in the replacement list and will be substituted by the actual macro argument upon invocation as described in the following section. The scope of each macro parameter extends from its appearance in the identifier list until the closing right parenthesis. There may be no duplicate macro parameter names in the identifier list.

After such a definition, subsequent occurrences of the named identifier that are followed by a parenthesis as the next unexpanded preprocessing token constitute an invocation of the function-like macro. The invocation will be substituted by the replacement list, after any argument substitution and token-pasting takes place, as described in the following section.

The tokens following the left parenthesis of the macro invocation, up to a matching terminating (unexpanded) right parenthesis constitute the macro argument list. Matching pairs of left and right parentheses may occur within a macro argument list.

Each argument within the macro argument list is separated by an unexpanded comma preprocessing token. Any commas that appear within string-literals, character-constants, or matching pairs of parentheses do not serve to separate macro arguments.

Within a macro argument list, all white space (including a new-line) is considered the same. This implies that preprocessing directives will not be recognized if they occur within a macro argument list. LPI-C will abide by this rule, although, strictly, the behavior in this situation is undefined and should not be relied upon.

If a macro argument consists of no preprocessing tokens (that it, if the macro argument is empty), then the behavior is undefined and should not be relied upon, although LPI-C will properly treat it as a null token and will issue a warning message.

If the number of macro arguments in the macro invocation does not agree with the number of macro parameters in the function-like macro definition, then the behavior is undefined and should not be relied upon. However, if too many arguments are given, LPI-C will discard any extra ones. If too few arguments are given, LPI-C will supply a null token for each missing argument.

Macro Argument Substitution

After each of the macro arguments of a function-like macro invocation has been collected, it is substituted for each occurrence of the corresponding macro parameter in the replacement list according to the following rules:

- If a macro parameter is preceded by a # preprocessing token, then the corresponding macro argument is the operand of the stringize operator. (The macro argument will first be expanded as described in the following section.)
- If a macro parameter is preceded or followed by a ## preprocessing token, then the corresponding macro argument is an operand of the token-paste operator and will first be expanded as described in the following section.
- Next, the macro argument corresponding to the macro parameter in the replacement list is itself fully expanded as if it formed the rest of the source file (that is, no other preprocessing tokens are available). Note that macro expansion can be a recursive process.

- Finally, the fully expanded macro argument is substituted for the corresponding macro parameter in the replacement list.

Macro Rescanning

Macro expansion is not yet finished after any token-pasting, stringizing, and argument substitution within the replacement list has taken place. The resulting sequence of preprocessing tokens is then rescanned for more object-like or function-like macro invocations to expand. In addition, the preprocessing tokens that make up the rest of the source file are available for expansion, unlike the case of macro expansion of macro arguments.

During the rescanning, if the name of the macro being replaced is found, it is not replaced and is no longer available for further replacement. Thus, "recursive death" is prevented and the kind of definition that follows is possible:

```
#define exit(status)  exit(status+1)
```

The preceding example will expand to the following:

```
exit(1+1)
```

If the final resulting sequence of preprocessing tokens has the same appearance as a preprocessing directive, it is not processed as though it were a preprocessing directive.

Function-Like Macro Examples

Function-like macros are generally used in the same way that functions are used. That is, they are used as a way of encapsulating some amount of computation, and, optionally, taking parameters or returning a value, or both.

The primary advantage function-like macros have over functions is speed. Because the macros are expanded at compile time to the specified code, they do not incur the extra overhead and processing associated with function calls.

A disadvantage of using function-like macros is that more code is produced if the macro is called more than once than would have been if a function were used. Also, function-like macros can be time-consuming to debug, while being very susceptible to programming errors. For example, if a macro parameter is referenced more than once, then the corresponding macro argument will be evaluated more than once. If the expression has a side-effect, then it is likely it will not be what was intended, as illustrated in the following example:

```
#define MIN(x,y)  (((x)<= (y)) ? (x):(y))

c = MIN(a,b) ;
c = MIN(a++,b++) ; /* DANGER! (a or b will be
                      incremented twice) */
```

Attention should also be given to properly parenthesizing macro parameters within the replacement list so that expressions are evaluated as they were intended. For example, the following expression

```
#define MUL(a,b) a * b
MUL(1+2,3);
```

will evaluate to "1 + 2 * 3" which is probably not what was intended. The following definition should be used instead to ensure the correct result:

```
#define MUL(a,b) ((a) * (b))
```

The # Operator

The # operator is known as the stringize operator, since it turns a sequence of tokens into a string-literal.

If a replacement list of a function-like macro definition contains a # preprocessing token that is immediately followed by a parameter, both the # preprocessing token and the corresponding macro argument are replaced by a single character string literal preprocessing token, whose spelling is identical to the spelling of the preprocessing token sequence for the corresponding argument.

The spelling of each preprocessing token in the argument prior to its replacement remains the same in the character string literal, with the following exceptions:

- white space that appears between preprocessing tokens in an argument becomes a single space character in the character string literal
- white space before the first preprocessing token is ignored
- white space after the last preprocessing token is ignored
- a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters)

The ## Operator

The ## operator is known as the token-paste operator, since it pastes two tokens into one.

If a replacement list of a macro definition contains a ## preprocessing token that is immediately followed or preceded by a parameter, the parameter is replaced by the preprocessing token sequence of the corresponding unexpanded macro argument.

Neither the object-like nor the function-like form of a macro definition may contain a ## preprocessing token at the beginning or the end of a replacement list.

Before the replacement list of either an object-like or a function-like macro invocation is re-examined for more macro names to replace, all ## preprocessing tokens in the replacement list are deleted.

Next, the preprocessing token that had preceded the ## preprocessing token is concatenated with the preprocessing token that followed it.

The result must be one and only one valid preprocessing token; otherwise, the behavior is undefined. Any resulting preprocessing token that is not valid will produce undefined behavior.

Removing a Macro Definition

The scope of a macro definition is in effect until a corresponding `#undef` directive. If there is no corresponding `#undef` directive, the macro definition scope will hold until the end of the compilation unit. (A macro definition scope is not affected by the beginning or end of a block, since the preprocessing phase has no notion of blocks.)

In the following expression, the preprocessing directive causes the identifier to be undefined (that is, removed) as a macro name.

```
# undef identifier new-line
```

If the identifier is not currently defined, the `#undef` directive is ignored. The following sequence illustrates the rules for redefinition and reexamination:

```
#define A       2
#define Q(x)    Q(A + (x))
#undef  A
#define A       3
#define R       Q
#define B       B[9]
#define S       S(!
#define T(f)    f(C)
#define C       4,5
#define U(a)    a
#define LP      (
#define RP      )
#define X       ,

Q(x - 1);
Q(Q(B));
U(U(R)(-1) / U) (-2);
R(A - Q + C) S 86);
T(Q);
T(T);
Q LP U(-9) RP;
Q(T LP A X B RP);
```

Yields:

```
Q(3 + (x - 1));
Q(3 + (Q (3 + (B[9]))));
Q(3 + (-1)) / U(-2);
```

```
Q(3 + (3 - Q + 4, 5)) S(!86);
Q(3 + (4, 5));
T(4, 5);
Q(-9);
Q(3 + (T(3, B[9])));
```

To illustrate the rules for creating character string literals and concatenating tokens, the following sequence:

```
#define NST(s)    #s        /* non-expanding */
#define NTP(a,b)  a##b      /* non-expanding */
#define ST(s)     NST(s)
#define TP(a,b)   NTP(a,b)
#define XPR(d)    pr(ST(TP(X,d)) ": %d\n",TP(X,d))
#define EX        h
#define H(s,v)    TP(TP(TP(TP(s,_),v),_),EX)
#define ZORBA     "Carpe"
#define BA        BA " diem"

#include ST(H(def,01))

pr (NST(H(def,01)) "=  %s\n",ST(H (def,01)));
XPR (offset);
pr (NST(f("Q\0;","Q",'\6') /* hi */
    !=0) NST(: @\n),z);
pr (NTP (ZOR, BA) "\n");
pr (TP (ZOR, BA) "\n");
```

Yields:

```
#include "def_01_h"  /* before file inclusion */

pr ("H(def,01)" "=  %s\n", "def_01_h");
pr ("Xoffset" ": %d\n", Xoffset);
pr ("f(\"Q\\0;\",\"Q\",'\\6') !=0" ": @\n",z);
pr ("Carpe" "\n");
pr ("Carpe" " diem" "\n");
```

Space around the # and ## tokens in the macro definition is optional.

Finally, to demonstrate the redefinition rules, the following sequence is valid.

```
#define MAX_SIZE    (1024 -1)
#define MAX_SIZE              (1024    -1)

#define MIN(a,b)    ((a) <= (b) ? (a) : (b))
#define MIN( a , b)      /**/((a) <= (b)/**/?/*
                           */(a) : (b))/**/

#define MAX(a,b)     ((a) >= (b) ? (a) : (b))
#define MAX(a , b)   ((a) <= (b) ? /* then */ \
                      (a)        : /* else */ \
                      (b))
```

The following redefinitions are invalid:

```
#define MAX_SIZE     1024
#define MAX_SIZE     (1024)

#define MIN(a,b)     ((a)<=(b)?(a):(b))
#define MIN(a,b)     ((a) <= (b) ? (a) : (b))

#define MAX(a,b)     ((a) >= (b) ? (a) : (b))
#define MAX(m,n)     ((m) >= (n) ? (m) : (n))
```

Conditional Compilation

This section deals with the manner in which specific sections of source files are selected to be included or excluded from processing, as determined by the chosen preprocessing directive.

In the case of the `#if` and `#elif` directives, the conditional inclusion or exclusion is based on whether a given controlling expression, which is an integral constant expression, evaluates to a non-zero value (true) or to a zero value (false). In the case of the `#ifdef` and `#ifndef` directives, the inclusion or exclusion is based on whether or not a specified name is defined or undefined as a macro.

Conditional Inclusion Directives

In the following preprocessing directives, **#if** and **#elif** test whether the controlling constant expression evaluates to nonzero.

#if *constant-expression new-line* $group_{opt}$
#elif *constant-expression new-line* $group_{opt}$

In the following preprocessing directives, **#ifdef** and **#ifndef** test whether or not the identifier is currently defined as a macro name.

#ifdef *identifier new-line* $group_{opt}$
#ifndef *identifier new-line* $group_{opt}$

In the case above, the first directive is equivalent to:

#if defined *identifier*

and the second directive is equivalent to:

#if !defined *identifier*

If the controlling expression of the directive evaluates to true (that is, non-zero), the associated section of source code will be included in the compilation. Only the first such group will be included (that is, in the case of **#elif** alternatives).

If the controlling expression is false (that is, zero), the associated section of source code will not be included in the compilation (that is, it will be excluded and will not be compiled).

If none of the conditions evaluates to true, and there is a **#else** directive, the **#else** is included in the compilation; if there is no **#else** directive, all the groups until the matching **#endif** are skipped.

Preprocessing tokens are not allowed between a **#else** or **#endif** directive and a terminating new-line character. That is, preprocessing tokens must appear by themselves on a line, although comments are allowed. For example:

```
#if XYZZY
#else XYZZY                    /* wrong */
#endif /* XYZZY */             /* okay  */
```

Also, since comments are processed in Phase 3 before the execution of preprocessing directives in Phase 4, they are significant within a skipped group. For example:

```
#if 0
this text is skipped
/* this is a comment; the following #else is...
#else
... commented out */
#endif
```

This example demonstrates the basic principle that preprocessing directives may be commented out.

Controlling Expressions

The expression that controls the inclusion or exclusion of specific sections of source code (that is, in a `#if` or `#elif` directive) must be an integral constant expression (as defined in Chapter 9). Note the following constraints:

- only integral constants or character constants may be used (no floating constants)
- the controlling expression may not contain a cast
- the controlling expression may contain unary operator expressions of one of the following forms:

  `defined` *identifier*

  or

  `defined (`*identifier*`)`

  These expression would then evaluate to 1 if the identifier is currently defined as a macro name, and 0 if it is not.

  (To be currently defined as a macro name, the identifier must be predefined, or be currently defined as a macro name by way of a `#define` preprocessing directive, as long as there is no intervening `#undef` directive.)

- Constants having type `int` act as if they had the same representation as `long`.

- Constants having type `unsigned int` act as if they had the same representation as `unsigned long`.
- Since all operands in the expression are either `long` or `unsigned long` integral constants, only operations that are allowed on these types are allowed.
- Since all operands in the expression are integral constants, the following operators cannot be used:
  - `sizeof` operator
  - increment (++) operators
  - decrement (--) operators
  - address of (&) operator
  - indirection (*) operator
  - structure/union pointer (->)
  - member (.) operator
  - array subscript ([ ]) operator
  - function call (( )) operator

Before the controlling expression of a `#if` or `#elif` directive is actually evaluated, any macro invocations are fully expanded. Macro names modified by the `defined` unary operator, however, are not expanded. Any identifiers remaining after macro expansion are replaced with the token `0` (zero).

Note
If the macro expansion process generates the `defined` token, the behavior is undefined. The behavior is also undefined if the `defined` unary operator was not used in one of the two specified ways.

Evaluation of the Controlling Expression

The controlling expression, consisting of the tokens resulting from macro expansion, is evaluated as an integral constant expression. This evaluation is performed according to the rules of constant expressions as described in Chapter 9.

When interpreting character constants in these constant expressions, escape sequences may be converted into execution character set members.

It is implementation-defined whether the numeric value for any character constant matches the value obtained when an identical character constant occurs in an expression that is not part of a `#if` or `#elif` directive.

For example, since there are distinct translation and execution environments, the constant expression in the following `#if` directive and `if` statement is not guaranteed to evaluate to the same value in both environments.

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

This behavior should not be relied upon. However, in LPI-C the evaluation of these two expressions is identical.

Line Numbering

When an error message is issued, LPI-C specifies the number of the line on which an error was encountered. That number is determined by presuming that the first line of the file is line 1 and incrementing the line numbers until the end of the file.

There are times when it may be useful to control the line number in a file by arbitrarily setting it to a particular line number.

For example, if the UNIX utility such as lex or yacc is used, the output may be a C source file which then may be included in the file.

If there is an error within that source file that was produced by lex or yacc, the error message would ascribe the error to a line number that is from that lex-generated source file, rather than to a line number from the original file.

To avoid this confusion, a `# line` preprocessing directive can be used to set a line number in the file being processed to the same number as in the lex-generated C source file.

The following preprocessing directive sets the line of a translation unit to a new number that is greater than zero and less than 32767.

`# line` *digit-sequence new-line*

The following preprocessing directive also sets the line number and changes the presumed name of the current source file to be the contents of the character string literal:

 # line *digit-sequence* "*s-char-sequence*$_{opt}$" *new-line*

The following preprocessing directive causes a replacement list to be replaced and then processes line numbering as in the two previously mentioned forms.

 # line *pp-tokens new-line*

Error Directive

Use of the following preprocessing directive form will produce a specific error message:

 # error *pp-tokens*$_{opt}$ *new-line*

This directive is commonly used to indicate an undefined conditional inclusion alternative. For example:

```
# if defined (m68_TARGET)
# define TARGET_IN "M68.in"
# elif defined (i386_TARGET)
# defined TARGET_IN "i386.in"
# else
# error "Only M68 or I386 targets supported"
# endif
```

Pragma Directive

Use of the following form of preprocessing directive will produce an implementation-defined result. (See Appendix C, item 13.v, in your *LPI-C User's Guide.*)

 # pragma *pp-tokens*$_{opt}$ *new-line*

Unknown **# pragmas** will be ignored.

Null Directive

Use of the following form of preprocessing directive has no effect.

```
# new-line
```

Predefined Macro Names

The following macro names are predefined by ANSI C:

__LINE__ Represents the line number of the source file the translator is currently processing. The line number is a decimal constant.

__FILE__ Represents the presumed name of the current source file as a character string literal.

__DATE__ Represents the date when translation of the source file occurred.

The translation date must be a character string literal of the following form:

Mmm dd yyyy

The names of the months are the same as those generated by the **asctime** function. If the day of the month is less than 10, then the first character of *dd* is a space character.

__TIME__ Represents the time when the source file translation occurred. The translation time is a character string literal of the following form:

hh:mm:ss

as generated by the **asctime** function.

__STDC__ LPI-C sets this to the decimal constant 1, which indicates a conforming implementation.

Predefined macros have values that stay constant until the end of the translation unit, with the exception of `__LINE__` and `__FILE__`.

These predefined macro names, as well as the `defined` identifier, cannot be the subject of a `#define` or a `#undef` preprocessing directive.

Chapter 5: Types

Chapter 5: Types

Overview

This chapter describes the data types supported by LPI-C. The type of the expression used to access a value from an object or a function determines the meaning of that value.

As described in Chapter 3, the simplest expression is an identifier. The particular type of the identifier is specified in the identifier's declaration.

Type Category

The broadest categories of types are the following:

- object types, which describe data objects
- function types, which describe functions
- incomplete types, which describe objects but do not contain the information needed to determine the object sizes

The type category of a derived type is its outermost derivation (see the section "Type Derivations" later in this chapter). The type category of a simple type is the type itself.

The following figure depicts the C data types and their relationships to each other.

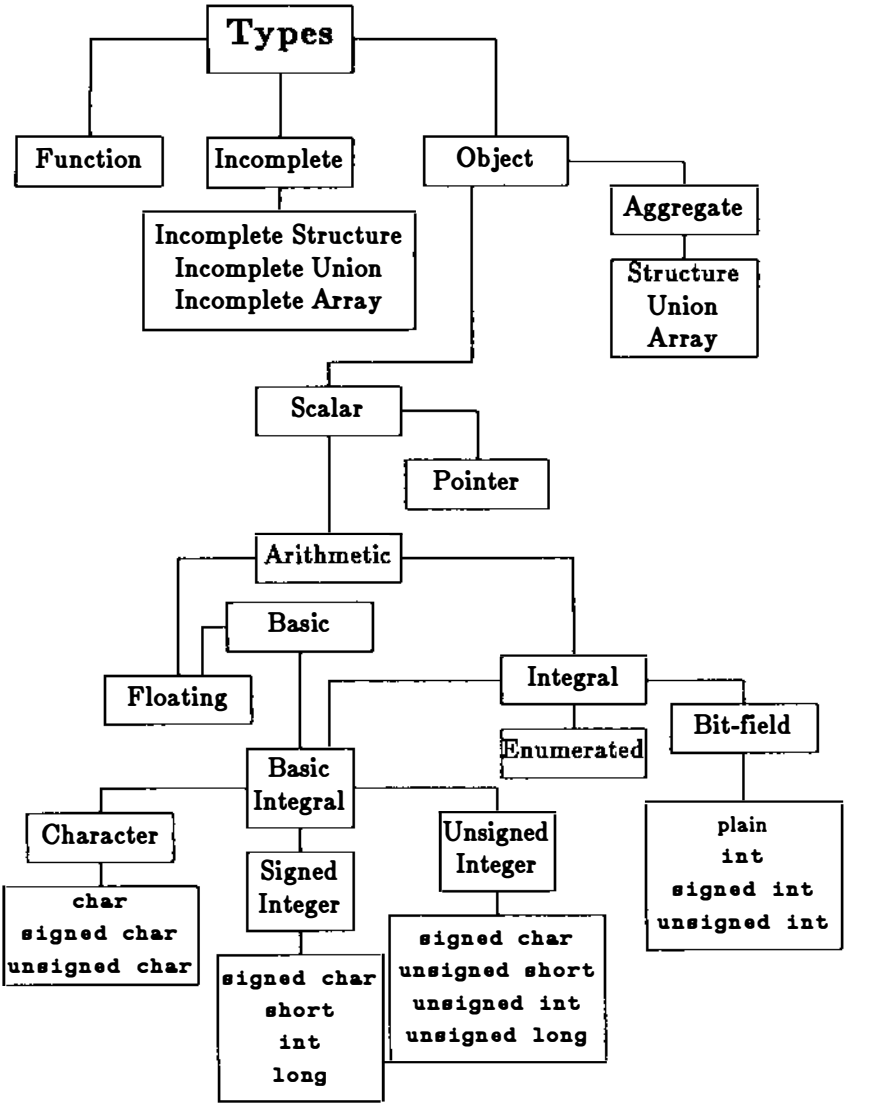


FIGURE 5-1 Types

Character Types

Character types include:

- `char`
- `signed char`
- `unsigned char`

Any object that has type `char` has enough space to store any member of the basic execution character set.

An object that has type `signed char` or `unsigned char` has the same amount of space as an object with type `char`. The way in which values outside the basic execution character set are stored in a character is implementation-defined. LPI-C will store the least significant part of the given value, as space provides (for example, the low order eight bits on most architectures).

LPI-C treats values stored in a "plain" `char` object as `signed`.

Signed Integer Types

There are four signed integer types:

- `signed char`
- `short int`
- `int`
- `long int`

A "plain" `int` object has sufficient space to contain a value that is in the range `INT_MIN` to `INT_MAX`, as specified in the header `<limits.h>`. (See Appendix D for more information.)

Of the signed integer types in the preceding list, `signed char` contains a subrange of the values of type `short int`; `short int` contains a subrange of the values of type `int`; and `int` contains a subrange of values contained in type `long int`. Note that in this case, the subrange may consist of the entire range of values of the next type in the list.

In addition, each of the signed integer types in the preceding list has a corresponding unsigned integer type that is designated with the keyword `unsigned`. Unsigned integer types use the same amount of space as signed integer types. Values of a signed integer type constitute a subset of the corresponding unsigned integer type, but still have the same alignment requirements.

There are several other ways in which the signed integer type, as well as other types, may be designated, as described in the section "Type Specifiers" in Chapter 7.

Floating Types

The three floating types are:

- `float`
- `double`
- `long double`

Of the floating types in the preceding list, `float` contains a subrange of the values of type `double`, and `double` contains a subrange of the values of type `long double`.

Enumeration Types

An enumeration consists of a list of identifier names, each of which has a distinct integer value. Each occurrence of an enumeration declares a distinct enumerated type.

In the following example, `color1` and `color2` are distinct enumerated types.

```
enum color1 {red, blue, white}
enum color2 {black, green, yellow}
```

Other Type Classifications

This section describes the other type classifications.

Basic Types

Basic types include:

- type `char`
- signed and unsigned integer types
- floating types

Although one of the basic types in the preceding list may be represented in the same way as another basic type, each distinct basic type is considered to be different from the others.

Arithmetic Types

Arithmetic types include the following:

- integral types
- floating types

Scalar Types

Scalar types include the following:

- arithmetic types
- pointer types

Aggregate Types

Aggregate types include the following:

- array types
- structure types

Because an object that has union type can only contain one member at a time, union types cannot be aggregate types.

Void Types

The type `void` is an incomplete type, containing no values, and cannot be completed.

Type Derivations

This section describes derived types, which are constructed from object, function, or incomplete types. Array, function, and pointer types are collectively called derived declarator types.

Structure Types

A structure type is a sequence of member objects. Those member objects may have different types and must have specified names (except for unnamed bit-fields).

Union Types

A union type is a sequence of member objects that overlap. Those member objects may have different types and must have specified names.

A structure or union type that has unknown content (as described in the section "Tags" in Chapter 7) is called an incomplete type. That structure or union type can be completed by declaring, later in the same scope, the same structure or union tag and defining the content.

Array Types

An array type is an ordered sequence of consecutively stored data objects. Each of those data objects has a specific type, called the element type. The number of objects in the sequence and the specific element type characterize an array type.

An array type is derived from the element type of the particular array. That is, the array type derivation is from its original type. If the element type of an array type is `int`, for example, then the array type is referred to as "array of `int`."

If an array type contains unknown size (for example, `char *av[ ]`), then it is called an incomplete type. That array type can be completed by specifying the size in a later declaration, which may have either internal or external linkage.

Function Types

A function is used to encapsulate the sequence of expressions and/or statements used to construct a program. Parameters may be passed to a function (by value) and a value may be returned from a function. The return type and the type and number of parameters characterize a function type.

A function type is derived from its return type and is referred to as the "function type derivation." If, for example, the return type is `int`, the function type can be referred to as "function returning `int`."

Pointer Types

A pointer type is an object that contains a value providing a reference to another type, called the referenced type. The derivation of a pointer type is from a referenced type and is referred to as "pointer type derivation." If, for example, a pointer type is derived from the referenced type `int`, it can be referred to as "pointer to `int`."

A pointer to void and a pointer to a character type have the same requirements for representation and alignment. Also, pointers to qualified versions of compatible types have the same requirements for representation and alignment as pointers to unqualified versions of the same compatible types.

However, pointers to other types do not need to have the same requirements for representation or alignment.

Qualified Types

All types described prior to this section are classified as unqualified types. Each unqualified type can be qualified in one of the following three ways:

- const-qualified
- volatile-qualified
- const-qualified and volatile-qualified

For example, the `int` is an unqualified type that has the three following corresponding qualified versions, each of which belongs to the same type category:

- `const int`
- `volatile int`
- `const volatile int`

Each of the qualified types also has the same requirements for representation and alignment, so that the types are interchangeable as function arguments, function return values, or union members. Even if a derived type is derived from a qualified type, that derived type is not itself qualified. For example, if a pointer type is derived from the type `const int`, then the resulting type `const int *` is itself not const-qualified; instead, it is a pointer to a const-qualified type.

Qualified Types Examples

The type "pointer to `int`" is designated as follows:

```
int *
```

The type category of the preceding example is not integer type but pointer type.

The const-qualified version of this type is designated as follows and is called "`const` pointer to `int`."

```
int * const
```

However, the type designated as follows is not a qualified type:

```
const int *
```

The type category of preceding example is a pointer to a qualified type and is called "pointer to const `int`."

The array type shown in the following example has a length of 10 and a function that has a single parameter of type `int`.

```
struct s1 (*[10])(int)
```

The type in the preceding example is called "array of pointer to function returning `struct s1`."

Compatible Types

Two types that have the same type are compatible types.

Note
Even if two types have the same representation, or even if two types are defined (by way of `typedef`) to be the same type, they are still not the same type and are thus not compatible. On the other hand, two types need not be identical to be compatible (for example, one type could be an incomplete version of the other type).

If two structure, union, or enumeration types declared in separate compilation units have the same number of members, as well as the same member names and compatible member types, then they are compatible.

Two structures must also have members in the same order to be compatible.

If two structures or unions have bit-fields, then they are compatible if the bit-fields have the same widths.

If two enumerations have members with the same values, then they are compatible.

If two or more declarations referring to the same object or function do not have compatible types, then the behavior is undefined. (See Chapter 7 for additional type compatibility rules relating to declarators, type specifiers, and type qualifiers.)

Composite Type

A composite type is a type constructed from two compatible types.

A composite type has the following characteristics:

- It is compatible with both of the types from which it is constructed.
- If one of the two types is an array of known size, then the composite type is an array of that size.
- If one of the two types is a function prototype (that is, a function type with a parameter type list) and the other is a function with an empty identifier list, then the composite type is a function prototype with the parameter type list.
- If, however, both of the two types are function prototypes (that is, function types with parameter type lists), then the composite type of the corresponding parameters in each parameter type list is used to construct the resulting composite parameter type list.

Any of the above-mentioned rules apply recursively to the types from which the composite type is constructed.

If an identifier in the same scope is redeclared with a compatible type, then the type of the identifier becomes the composite type of each declaration.

Composite Type Example

Given the following two declarations with the same scope:

```
int func(float (*) [2], char (*) ());
int func(float (*) [], char (*) (int *));
```

The resulting composite type for the function is:

```
int func(float[*][2], char (*) int (*));
```

Chapter 6: Scope

Chapter 6: Scope

Overview

This chapter describes the different scopes, name spaces, and linkages of identifiers, as well as the storage durations of objects and external object definitions.

In addition, this chapter describes external and function definitions.

Scopes of Identifiers

The scope of a program is the region of program text in which an identifier is visible. Outside of the scope, the identifier is not recognized.

If two identifiers with the same name exist in the same name space, then the outer declaration of that identifier is superseded by the inner declaration until the inner scope terminates. (That is, the outer declaration is effectively "masked out" within the inner scope declaration.)

There are four kinds of scope:

- function
- file
- block
- function prototype

If the scopes of two identifiers end at the same point, then those identifiers have the same scope.

The scope of an identifier begins just after its declarator is completed, with the following exceptions:

- the scopes of structure, union, and enumeration tags begin just after the tag in the type specifier in which the tag is declared

- the scope of an enumeration constant begins just after its defining enumerator in an enumerator list

Function Scope

Only a label name, followed by a colon (:) and a statement, has function scope. The label name must be unique within a function and can be used (in a `goto` statement) anywhere within that function.

File Scope

An identifier has file scope if its declaration is outside any block and is not part of a parameter declaration. File scope ends at the end of the file.

Block Scope

An identifier has block scope if its declaration is inside a block or within an old-style function definition's list of parameters. Block scope ends at the right brace (}) that terminates the block.

Function Prototype Scope

An identifier has function prototype scope if its declaration is within the list of parameter declarations in a function prototype declaration which is not part of a function definition. Function prototype scope ends at the end of the function declarator.

Name Spaces of Identifiers

At any point in the translation unit, an identifier may be encountered that has more than one visible declaration associated with it. Although this may at first seem to present an ambiguous situation, it does not; the syntactic context in which the identifier appears is used to determine which particular declaration (and thus which entity) the reference refers to. This process of disambiguation leads naturally to the formation of a number of categories of identifiers, called "name spaces."

The following list outlines name spaces for the different identifier categories:

- Label names, distinguished by the syntax of the label declaration and use.
- Tags of structures, unions, and enumerations (there is one name space for all three entities); these identifiers are distinguished from other identifiers by the tags following any of the keywords `struct`, `union`, or `enum`, respectively).
- Members of structures or unions; they are distinguished by the type of the expression used to access the member by way of the `.` or `->` operator.
- Ordinary identifiers, which are all other identifers. These identifiers can be declared in ordinary declarators or as enumeration constants.

Linkages of Identifiers

By the use of linkage, an identifier that is declared in more than one scope, or in the same scope more than once, can be forced to refer to the same object or function.

The three kinds of linkage are as follows:

- external
- internal
- none

The kind of linkage an identifier has depends on one or more of the following:

- the storage class with which the declaration was made
- the scope in which the declaration was made (that is, where the declaration was made)
- whether the identifier is an object or a function type

Within an entire program, each time an identifier with external linkage appears, that identifier refers to the same object or function. Within a single translation unit, each time an identifier with internal linkage appears, that identifier refers to the same object or function. An

identifer with no linkage refers to a single object or function. (Within a translation unit, an identifier that appears with both internal and external linkage will result in undefined behavior.)

The linkage of an object or a function declared with the `static` storage-class specifier at file scope is internal (but if it is declared at block scope, it has no linkage).

The linkage of an object or a function declared with the storage-class specifier `extern` is the same as any visible file scope declaration of that identifier. In the absence of a visible file scope declaration, the object or function has external linkage.

The linkage of a function declared with no storage-class specifier is the same as if it were declared with the `extern` storage-class specifier. The linkage of an object (as opposed to a function) declared with no storage-class specifier at file scope is external.

An identifier has no linkage if it is declared to be one of the following:

- anything other than an object or a function (for example, a `typedef` name; structure, union, or enum tag; enumeration constant; or label name)
- a function parameter
- a block scope identifier, declared for an object that does not have the storage-class specifier `extern`

The following table summarizes linkage of file and block identifiers.

TABLE 6-1 Scope Linkage

STORAGE-CLASS	FILE SCOPE	BLOCK SCOPE
None	If the identifier being declared is of type function then same as `extern` with file scope (below) otherwise, external linkage	No linkage
`extern`	Same linkage as any visible file scope declaration of the identifier; if none, then external linkage	Same as `extern` file scope
`static`	Internal linkage	No linkage

Note that:

- Function scope identifiers (that is, label names) have no linkage.
- Prototype scope identifiers (that is, parameters; structure, union, or enum tags; or enumeration constants) have no linkage.

Storage Durations of Objects

The storage duration of an object determines that object's lifetime. Storage duration can be static or automatic.

Static Storage Duration

An object has static storage duration when its identifier is declared with external or internal linkage or is declared with the storage-class specifier `static`.

An object with static storage duration has its storage reserved and its stored value initialized prior to program startup. The last-stored value

of that object will be retained until the end of the execution of the program. (The last-stored value of a `volatile` object may not be explicit in the program; see Chapter 7.)

Automatic Storage Duration

An object with automatic storage duration is an object whose identifier is declared without the storage-class specifier `static`, and with no linkage (this implies block scope). An object with automatic storage duration is guaranteed to have memory allocated to it each time the block in which the object is declared is entered normally. Alternatively, that object will have memory allocated to it each time the block is entered by a jump to a labelled statement within the block (or an enclosed block) from outside the block.

If the value stored in the object is to be initialized upon entry into the block, then that initialization is performed on each normal entry. However, that initialization is not performed if the block is entered by a jump to a labeled statement within the block.

When execution of the block ends, even if not normally (for example, by way of `goto` or `break`), the reserved storage for the object is no longer guaranteed. (The value of a pointer that referred to an object whose reserved storage block has gone out of scope is undefined.)

The process of entering an enclosed block and calling a function from within the block will suspend but not end the execution of the enclosing block.

External Definitions

A compilation unit is the unit of program text that results after preprocessing (see Chapter 1). A compilation unit is a sequence of external declarations (note that they are called external because they appear outside any function, not because of anything related to the `extern` storage-class specifier). External declarations therefore have file scope. (The declaration specifiers of an external declaration cannot contain the storage-class specifiers `auto` and `register`.)

A definition is a kind of declaration that also causes storage to be reserved for the object or a function named by the identifier.

When an identifier declared with external linkage is part of an expression (unless that identifier is part of the operand of a `sizeof` operator), at some location in the program there must be one and only one external definition for the identifier. Otherwise, the behavior is undefined. There does not need to be a corresponding external definition for an identifier that is declared with external linkage and is not used in an expression (or is used only within a `sizeof` expression).

Only one external definition may appear for each identifier in the translation unit with internal linkage. In addition, if an expression contains an identifier declared with internal linkage (unless the identifier is part of the operand of a `sizeof` operator), there must be one and only one external definition in the translation unit for the identifier.

Translation Unit Syntax

translation-unit:
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

Function Definitions

A function definition contains a function declarator, which provides both the name of the function and its parameter identifiers and types, if any.

The declarator may include a parameter type list, which provides information describing the number, types, and possibly the names of the function parameters. A function declarator that includes a parameter type list is said to be a "prototyped" function declarator.

A declarator that does not include a parameter type list (that is, is not prototyped) may include an identifier list that may be empty. If that identifier list is not empty, then the following declaration list will specify the types of the parameters named in the identifier list, which may be used in the function body that must follow. An undeclared parameter is assumed to have type `int`. Such a function declarator is said to be an "old-style" function declarator.

When a prototyped function is entered (after being called), each of its arguments is converted, as if by assignment, to the type of the corresponding parameter.

If a parameter is declared as array of type, it is adjusted to pointer to type. If a parameter is declared as function returning type, it is adjusted to pointer to function returning type (see the section "lvalues and Function Designators" in Chapter 8).

All parameters have automatic storage duration; their storage layout is unspecified.

Because the effect of the parameter declaration is to define the parameter at the head of the compound statement that constitutes the function body, a parameter cannot be redeclared in the function body (unless it is enclosed within an inner block.)

A function may return type `void`, or any object type, except an array type.

The only storage-class specifiers allowed to be used in a function declarator (or definition) are `extern` or `static`.

The only storage-class specifier allowed to be used in a function parameter declaration (either in a prototyped parameter type list or an old-style declaration list) is `register`.

For a prototyped function definition, each parameter declaration must include an identifier and must not be followed by an "old-style" declaration list. The exception to this rule is the special case of a parameter list containing a single parameter of type `void`; no identifier should be specified for `void`.

For an old-style function definition, the parameter declaration list may declare only identifiers named in the identifier list.

The identifier in a function definition (that is, the function name) cannot inherit its type information from a `typedef`. For example,

```
typedef int F (char);
F function { };
```

is illegal, but

```
F func;
int func (char) { };
```

is correct since `F` and `func` have compatible types.

If an identifier has been declared as a `typedef` name, then it may not be redeclared as a parameter.

Function Definition Syntax

function-definition:
declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$
compound-statement

Function Definition Examples

In the following function definition, the storage-class specifier is `extern` and the type specifier is `int`. Both `extern` and `int` are the defaults and could thus be omitted.

```
extern int sum (int i, int j)
{
    return (i + j);
}
```

In the preceding prototyped definition, the function declarator is `sum(int i, int j)` and the function body is the following:

```
{ return (i + j};
```

In the following old-style definition, the definition declares its parameters by way of the identifier-list and a following declaration list. The declaration list for the parameters is `int i, j;`. Because `i` and `j` are both of type `int`, their declaration may be omitted, since that is the default type.

```
extern int sum(i, j)
int i, j;
{
    return (i + j);
}
```

A prototyped function definition forces conversion of the arguments of subsequent calls to the function, as if by assignment to the type of the corresponding parameter. Old style function definitions, on the other hand, do not force argument conversions (the default argument promotions are simply performed on each argument).

The following example illustrates the passing of one function to another.

```
int f(char);
/*...*/
g(f);
```

A prototyped definition of `g` might then follow as:

```
g(int (*fp)(char))
{ char c;
    (*fp)(c)            /* or */
    fp (c);
}
```

External Object Definitions

An external object definition is the declaration of an identifier for a data object, when that declaration has file scope and an initializer.

A tentative external object definition is a declaration of an identifier for an object, when that declaration has file scope but does not have an initializer; furthermore, it may not have a storage-class specifier other than `static`.

If there are one or more tentative definitions for an identifier in the translation unit and there is no external definition for that identifier, then the compiler will behave as if the following were true:

- a file scope declaration is present
- as of the end of the compilation, the identifier has composite type
- the declaration contains an initializer that is equal to 0

The declared type cannot be an incomplete type if the declaration is a tentative definition and has internal linkage.

External Object Examples

In the following examples, assume external and internal definitions as noted.

```
int e1 = 1;                 /* external definition */
extern int e2 = 2;          /* external definition */
int e3;                     /* external tentative definition */
static int i1 = 1;          /* internal definition */
static int i2;              /* internal tentative definition */
```

Then, the following are valid tentative definitions, each referring to its corresponding previous definition:

```
int e1;                     /* valid tentative definitions */
int e2;                     /* valid tentative definitions */
int e3;                     /* valid tentative definitions */
```

but the following have a linkage conflict between external and static:

```
int i1;                 /* extern: previously declared internal */
int i2;                 /* extern: previously declared internal */
```

Chapter 7: Declarations

Chapter 7: Declarations (Cont.)

Chapter 7: Declarations

Overview

This chapter describes declarations, which specify the interpretation of identifiers.

A declaration does not necessarily reserve storage for an object or function named by an identifier. A declaration that also reserves storage is called a definition.

Declaration Syntax

declaration:
declaration-specifiers *init-declarator-list*$_{opt}$;

declaration-specifiers:
storage-class-specifier *declaration-specifiers*$_{opt}$
type-specifier *declaration-specifiers*$_{opt}$
type-qualifier *declaration-specifiers*$_{opt}$

init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*

init-declarator:
declarator
declarator = *initializer*

Declaration Components

A declaration must consist of at least one of the following:

- a declarator
- a tag (structure, union, or enumeration)
- the members of an enumeration

An *init-declarator-list* is a sequence of declarators that are separated by commas. Those declarators contain the identifiers being declared.

A declarator in the *init-declarator-list* may have the following:

- an initializer
- additional type information
- both an initializer and additional type information

A declaration specifier contains a sequence of specifiers that interpret the linkage, storage duration, and type of the identifier.

If an identifier has no linkage, then there can only be one declaration of that identifier that has the same scope and the same name space. The only exception to this rule is the case of tags (see the section "Tags" later in this chapter).

If an identifier has no linkage for an object, then by the end of its declarator, the type of that object must be complete. If that identifier has an initializer, then its type must be complete by the end of its *init-declarator*.

If two or more declarations refer to the same object or function in the same scope, then they must have compatible types.

Storage-Class Specifiers

Storage-class specifiers (as listed in the section "Storage-Class Syntax") give storage class attributes to declared objects. No more than one storage-class specifier may be given in a declaration.

Although not required, it is general practice to place the storage-class specifier at the beginning of the declaration specifiers.

`typedef`

For discussion on use of the `typedef` specifier, see the section "Type Definitions" later in this chapter.

extern

When the storage-class specifier **extern** is used within a block, it specifies that the object is defined elsewhere; that is, it is defined elsewhere in the same compilation unit or in another compilation unit.

For a description of how **extern** is used outside a function, see Chapter 6.

If an identifier is declared for a function with block scope, then the only storage-class specifier that can be used is **extern**. If no storage-class specifier is used, then **extern** is assumed.

static

When the **static** storage-class specifier is used inside a block, it defines the storage of the given identifier to be local to that block. (That is, its scope is restricted to that block.)

When **static** is used at file scope level, it represents a declaration or the definition of a function and has internal linkage. (See Chapter 6 for further discussion of file scope.)

The duration of a static object remains active throughout execution of the entire program.

auto

The **auto** storage-class specifier gives automatic storage class to declared objects and can only be used in functions.

The duration of such an object is limited to the active invocations of its containing function.

register

The **register** specifier can only be used in functions, like the **auto** storage-class specifier. The **register** specifier is used when access to the object will be frequent and needs to be done quickly.

The compiler may ignore a **register** declaration and treat it simply as an **auto** declaration. The actual number of declarations for which this request is honored by the compiler is implementation-defined. (See Appendix C, item 8.i, in the *LPI-C User's Guide.*)

The contents of a **register** variable are not addressable, even if the compiler chooses to treat a **register** declaration as an **auto** declaration. Thus, the address-of operator may not be applied to a **register** variable, nor may an array with a register specifier be converted to a pointer. The only operator that may be applied to such an array is **sizeof**.

Storage-Class Syntax

storage-class-specifier:
typedef
extern
static
auto
register

Type Specifiers

Type specifiers may occur in any order and can be separated from each other by other declaration specifiers, such as storage-class specifiers.

Within a declaration specifier, all type specifiers must be from one of the sets of specifiers in the following list, in which each set of type specifiers is delimited by commas to indicate alternative ways of specifying the same type. (As an exception, for bit-field declarations, whether the type **signed int** or **signed** differs from **int** is implementation-defined. LPI-C treats a "plain" **int** bit-field as an **unsigned int**.)

- **void**
- **char**
- **signed char**
- **unsigned char**
- **short**, **signed short**, **short int**, or **signed short int**

- **unsigned short**, or **unsigned short int**
- **int**, **signed**, **signed int**, or no type specifiers
- **unsigned**, or **unsigned int**
- **long**, **signed long**, **long int**, or **signed long int**
- **unsigned long**, or **unsigned long int**
- **float**
- **double**
- **long double**
- *struct-or-union specifier*
- *enum-specifier*
- *typedef-name*

Type Specifier Syntax

type-specifier:
void
char
short
int
long
float
double
signed
unsigned
struct-or-union-specifier
enum-specifier
typedef-name

Structure and Union Specifiers

Structure and union specifiers have the same form. However, a structure and a union are distinguished in the following ways:

- A structure is an object that consists of a sequence of named members, each of which can be of any type. The members of a structure are stored in an ordered sequence.
- A union is a type consisting of a sequence of named members. Members in a union have overlapping storage.

The structure or union may contain a *struct-declaration-list*, which is a sequence of declarations for the members of that structure or union.

If a *struct-declaration-list* appears in a *struct-or-union-specifier*, then within the translation unit a new type is declared. The list terminates with a right brace (}), and until that brace is reached, the type is incomplete.

The behavior is undefined when there are no named members for the *struct-declaration-list.*

Bit-Fields

A bit-field is a member of a structure or union whose field width consists of a specified number of bits, one of which can be a sign bit. A bit-field must have type `int`, `unsigned int`, or `signed int`.

The width of a bit-field is represented in the structure or union syntax by the constant expression following the colon, as shown in the following example:

```
struct {
     int is_1_bit              : 1;
     unsigned int is_2_bits    : 2;
     signed int is_3_bits      : 3;
} flags;
```

The number of bits in a bit-field may not exceed the number of bits in an `int`. Whether the high-order bit position of a "plain" `int` bit-field is treated as a sign bit is implementation-defined. LPI-C will treat "plain" `int` bit-fields as unsigned. A bit-field is interpreted as an integral type consisting of the specified number of bits.

ANSI C allows the implementation to allocate any storage unit large enough to contain the bit-field. (See Appendix C, item 9.v, in the *LPI-C User's Guide.*) Extra storage space that is not needed for the bit-field can then be used by an immediately following bit-field, if there is one.

If, however, there is not enough space remaining for the next bit-field, then ANSI C allows the implementation either to place any superfluous bit-field members into the following units, or to put the entire bit-field into the following unit. LPI-C will do the latter; that is, LPI-C will not allow a bit-field to straddle a storage unit boundary; a bit-field will always be combined completely within its storage-unit.

The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. LPI-C will pack bit-fields into `int`s from the most significant bit toward the least significant bit.

If a bit-field declaration has only a colon and a width, but no declarator, then the bit-field is unnamed.

If, however, a bit-field defines its field width to be 0, then no further bit-field members can be packed into the storage unit in which the previous bit-field, if any, was placed; that is, a zero length bit-field specifies that the next bit-field members (if any) should start at the beginning of the next storage-unit.

ANSI C allows the implementation to define how to align a member of a structure or union object that is not a bit-field. (See Appendix C, item 9.ii, in the *LPI-C User's Guide* for further information.)

LPI-C will place each member at a byte offset (from the beginning of the structure or union) which is an integral multiple of the alignment requirements for its data type; the alignment for an aggregate type is defined to be the most stringent alignment requirement of any of its members or elements.

The address-of operator (&) cannot be used with bit-field objects. Therefore, there can be no pointers to or arrays of bit-field objects (an array of structures containing bit-fields can, of course, be defined).

Within a structure object, non-bit-field members and the storage units in which bit-fields reside have addresses which increase in the order in

which they are declared. A pointer to a structure object, converted to a pointer to the type of its initial member (that is, by way of a cast), will point to its initial member and vice versa.

There may be unnamed holes within a structure object, to achieve the appropriate alignment, but not at the beginning of the structure.

The size of a union is large enough to contain the largest of its members. The value of no more than one of the members of a union may be stored in a union object at any time. A pointer to a union object, converted to a pointer to the type of any of its members (that is, by way of a cast), will point to that member and vice versa.

There may be unnamed padding at the end of a structure or union, to achieve the appropriate alignment if the structure or union is an element of an array.

A structure or union may not contain a member with incomplete or function type. It may, therefore, not contain an instance of itself. (It may, however, contain a pointer to an instance of itself.)

The expression that specifies the width of a bit-field must be an integral constant expression that has nonnegative value that does not exceed the number of bits in an ordinary object of compatible type. If the value is zero, the declaration has no declarator.

Structure and Union Syntax

struct-or-union-specifier:
struct-or-union identifier$_{opt}$ { *struct-declaration-list* }
struct-or-union identifier

struct-or-union:
`struct`
`union`

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

struct-declaration:
specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
struct-declarator
struct-declarator-list , struct-declarator

struct-declarator:
declarator
declarator$_{opt}$ *: constant-expression*

Enumeration Specifiers

An enumeration list contains identifiers that are declared as constants with type `int`. Within the same scope, each enumeration constant is different from other enumeration constants and is also different from other identifiers.

As shown in the following example, an enumerator (also known as a member of an enumeration) containing the symbol = gives its enumeration constant the value of the constant expression that follows.

enumeration-constant = *constant-expression*

However, if the first enumerator in the declaration does not contain the symbol =, then the enumeration constant is assigned the value of 0.

If any of the enumerators that follow the first enumerator do not contain the symbol =, then each enumeration constant is defined by adding 1 to the value of the previous enumeration constant.

Note that when the symbol = is used within an enumerator, the resulting values of some enumeration constants may be the same as other values in the same enumeration.

This constant expression must be an integral constant expression with type `int`. ANSI C allows the implementation to define the enumerated type, but it must be compatible with an integer type. LPI-C defines each enumerated type to be type `int`.

Enumeration Specifier Syntax

enum-specifier:
  enum *identifier*$_{opt}$ { *enumerator-list* }
  enum *identifier*

enumerator-list:
  enumerator
  enumerator-list , *enumerator*

enumerator:
  enumeration-constant
  enumeration-constant = *constant-expression*

Enumeration Constant Example

In the following example, the tag of the enumeration is **days**. (See the following section "Tags.")

The enumeration constant then declares that the object **work_day** is of type **days** and that **wd** is a pointer to an object with type **days**. The following set contains the enumerated values: { 0, 1, 18, 19, 20}.

```
enum days { mon, tues, wed=18, thurs, fri };
/*...*/
enum days work_day, *wd;
/*...*/
work_day = tues;
wd = &workday;
/*...*/
/*...*/ (*wd != fri) /*...*/
```

Tags

A tag is a type specifier that can be used to mark a structure, union, or enumeration for later use.

In the following form, the identifier serves as the tag. A subsequent declaration can then refer to the tag and omit the bracketed list, as long as the declaration is contained in the same scope.

struct-or-union identifier { *struct-declaration-list* }

or

enum *identifier* { *enumerator-list* }

A structure or union declaration with a tag may appear anywhere in a compilation unit, as follows:

struct-or-union identifier

However, if the tag appears before the contents of the structure or union are defined, then the structure or union is an incomplete type. To complete the type, the tag must be later defined within the same scope.

The tag of an incomplete type, as above, may be used only when the size of an object of the specified type is not needed. The size of the object is not needed, for example, when a **typedef** specifier is declared for a structure or union, or when declaring a pointer to or a function returning a structure or union. (See the section "Incomplete Types" in Chapter 5.)

The following construct declares a structure or union and associated tag that is local to its own scope and specifies a type that is different from other types having the same tag in the enclosing scope. Both the type and the tag are in effect only within the scope of the declaration.

struct-or-union identifier ;

The following construct specifies a new structure, union or enumeration.

struct-or-union { *struct-declaration-list* }

or

`enum` { *enumerator-list* }

Since a tag is not part of the declaration, the type cannot be referred to outside of its own declaration (except when it is the declaration of a `typedef` name).

Tag Declaration Examples

In the following example, a tag allows the declaration of a self-referential structure.

The structure specified is called `link`; `link` contains an integer and two pointers to objects of `link` type. (Because the pointers within the declaration also point to the type that is specified by the declaration, it is a self-referential structure.)

```
struct link {
    int info;
    struct link *next, *prev;
};
```

Now that the structure type `link` has been declared, it can be referred to elsewhere in the program in the following way:

```
struct link l, *lp;
```

In this example, `l` is declared as an object of `link` type and `lp` as a pointer to an object of `link` type.

Alternatively, the `typedef` mechanism can be used to produce the same results, as in the following example:

```
typedef struct link LINK;
struct link {
    int info;
    LINK *next, *prev;
};
LINK l, *lp;
```

Tag declaration can also be used to declare two mutually-referential structures. The following example contains two structures that contain pointers to each other.

```
struct s1 { struct s2 *p2; };
struct s2 { struct s1 *p1; };
```

However, if, in the preceding example, the tag `s2` had already been declared within the same scope, then the first declaration of `s1` would refer to it rather than to its forward declaration as shown above.

To avoid the possibility of confusion, the incomplete declaration

```
struct s2;
```

should be used before the declaration of `struct s1`.

This declaration will override any earlier versions of the tag `s2` in a containing scope and serve to properly complete the declaration of `s1`.

Type Qualifiers

The unqualified type of an lvalue can be qualified as `const`, `volatile`, or as both `const` and `volatile`. (For further discussion of qualified types, see Chapter 5.)

LPI-C will always provide storage for a `const`-qualified type, regardless of whether it is actually referred to in the program. LPI-C will, in most cases, detect direct modifications of const-qualified types at compile time.

Behavior is undefined in the following circumstances:

- when an lvalue with non-`const`-qualified type is used to modify an object that is defined with a `const`-qualified type
- when an lvalue with non-volatile-qualified type is used to refer to an object defined with a volatile-qualified type

An object with `volatile`-qualified type may be modified by factors unknown to the compiler (for example, an object at a memory-mapped input/output address may be affected by external programs). The compiler will evaluate an expression containing a `volatile`-qualified type according to the rules of the abstract machine. (See Chapter 2.)

Except for those modifications unknown to the compiler, the value stored in the volatile-qualified object at any given sequence point will be the same as the value required by the semantics of the abstract machine. LPI-C will not perform optimizations on a `volatile` object which violates these rules.

In an array type that contains type qualifiers, it is the element type that is qualified, not the array type.

Type qualifiers can only be present in a function type or an array type through the use of `typedefs`. In the case of a function type, however, the behavior is undefined.

Qualified types are compatible when they have the identically qualified version of a compatible type. Within a list of specifiers or qualifiers, the order in which type qualifiers appear will not affect the specified type.

Type Qualifier Syntax

type-qualifier:
`const`
`volatile`

Type Qualifier Examples

The object declared in the following example is a "read-only" external variable and cannot be modified in any way except by the hardware (that is, it cannot be assigned to, incremented, or decremented).

```
extern const volatile int read_only_extern;
```

When type qualifiers modify an aggregate type, the resulting behavior is as shown in the following examples:

```
const struct s { short i; } cs = { 10 };
struct s mcs;  /* the object mcs is modifiable */
typedef long int A[2][3];
const A a = {{1,2,3}, {4,5,6}};  /* array of array
                                     of const long */
short *pi;
const short *pci;


mcs = cs;        /* okay */

cs = mcs;        /* wrong:  cs is const */

pi = &mcs.i;     /* okay */

pi = &cs.i;      /* wrong:  &cs.i is ptr-to-const-
                            short */

pci = &cs.i;     /* okay */

pi = a[0];       /* wrong: a[0] is ptr-to-const-
                             int */
```

Declarators

A declarator declares an identifier, which, when used as an operand within an expression, designates an object (or function) having the same specifications as appearing in the declaration (that is, scope, storage duration, and type).

Declarator Syntax

declarator:
    pointer$_{opt}$ *direct-declarator*

direct-declarator:
    identifier
    (*declarator*)
    direct-declarator [*constant-expression*$_{opt}$]
    direct-declarator (*parameter-type-list*)
    direct-declarator (*identifier-list*$_{opt}$)

pointer:
    * *type-qualifier-list*$_{opt}$
    * *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier

parameter-type-list:
    parameter-list
    parameter-list , ...

parameter-list:
    parameter-declaration
    parameter-list , *parameter-declaration*

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
    identifier
    identifier-list , *identifier*

In the following declaration:

```
T D1
```

- T contains the declaration specifiers that specify what the type T is (such as `int`)

- D1 is a declarator that contains an identifier `x`

The following illustrates the type specified for the identifier `x` in the different forms of declarator:

- Assume that in the declaration "T D1," D1 has the following form:

  identifier

  Then, the type specified for `x` is T.

- Assume that in the declaration "T D1," D1 has the following form:

  `( D )`

  Then, `x` has the type specified by the declaration "T D."

Although a declarator in parentheses is the same as the unparenthesized declarator, complex declarators may have their meanings altered by the presence of parentheses.

ANSI C requires that the implementation allow types to be specified with at least 12 pointer, array, and function declarators, combined in valid ways, that will either directly modify an arithmetic, structure, union, or incomplete type, or will modify those types by way of one or more `typedefs`. LPI-C allows up to 15 such combinations.

Pointer Declarators

Assume that in the following declaration

T D1

T represents the type, D represents the declarator, and D1 has the following form:

$$* \textit{ type-qualifier-list}_{opt} \text{ D}$$

In the declaration "T D," the type specified for `x` is

"derived-declarator-type-list T"

and the type specified for **x** is:

"derived-declarator-type-list type-qualifier-list pointer to T."

Then, **x** is a pointer, qualified by each type qualifier in the list.

Two pointer types are compatible only when they are pointers to compatible types and when they are qualified in the same way.

Pointer Examples

The following declaration illustrates a variable pointer to a constant value, in which the contents of an object pointed to by `ptr_to_constant` cannot be changed through that pointer. However, the pointer itself may be modified to point to another object.

```
const int *ptr_to_constant;
```

The following declaration illustrates a constant pointer to a variable value. The contents of the `int` pointed to by `constant_ptr` may be changed. However, `constant_ptr` will always be a pointer to the same object.

```
int *const constant_ptr;
```

In the following example, a definition for the type "pointer to int" is added in order to further clarify the declaration of `constant_ptr`.

```
typedef int *int_ptr;
const int_ptr constant_ptr;
```

Array Declarators

In the following declaration,

T D1

assume T represents the type, D1 represents the declarator, and D1 has the following form:

$D[constant\text{-}expression_{opt}]$

If, in the declaration T D, the type specified for `x` is:

derived-declarator-type-list T

Then, the type specified for `x` is:

derived-declarator-type-list array of T.

An array may be constructed from an arithmetic type, a pointer, a structure or union, or from another array. (If an array is constructed from another array, a multi-dimensional array will be generated.)

The size of the array is specified by an integral constant expression delimited by `[` and `]`; that expression has a value greater than zero.

If the size of the array is not given, then the array type is an incomplete type. If the sizes are specified for both array types, then they must be of the same value.

If two array types are compatible, then their element types must be compatible.

Array Examples

The following expression declares an array of `doubles`, followed by an array of pointers to `doubles`, and a pointer to an array of `doubles`.

```
double da[10], *adp[10], *(*padp)[10];
```

The following example declares `x` as an array of `int` of unspecified size, and therefore an incomplete type. The storage for `x` must be defined elsewhere, possibly in another translation unit.

```
extern int x[];
```

Function Declarators

If we consider the following declaration

T D1

where `D1` has one of the two following forms

D(*identifier-list*$_{opt}$)

or

D(*parameter-type-list*)

and the type specified for `f` in the declaration T D is the following:

derived-declarator-type-list T

then the type specified for `f` is

derived-declarator-type-list function returning T

The first of the preceding forms specifies an old-style function declarator. This function form, still supported by ANSI C, can contain an optional identifier list (if part of an old-style function definition), or can consist of simply an empty set of matching parentheses (if it is part of an old-style function declaration, or an old-style function definition taking no arguments).

If the function declarator is part of a function definition, then the identifier list declares only the names of the function parameters. An empty list in a function declarator that is part of a function definition specifies that the function has no parameters.

If the function declarator is not part of a function definition, the empty identifier list specifies that the number or types of the parameters is unknown.

In the second of the preceding forms, the prototyped function declarator includes a parameter type list, which specifies the number and types of arguments of a function. It must also declare the names of the function parameters if it is part of a function definition. Otherwise, if it is only a function declaration, then it may declare the names of the function parameters, but is not required to. If present, those parameter names will be ignored.

A parameter list ending with an ellipsis (, . . .) indicates that the function may take a variable number of arguments and provides no information about the number or types of the parameters that appear after the comma in the list. (However, the macros defined in the

`<stdarg.h>` header may be used to access arguments that correspond to the ellipsis in a portable manner. See your *LPI-C Library Reference Manual.*)

The presence of `void` as the only item in the parameter list specifies that the function has no parameters. Note that the parameter type list (`void, . . .`) specifies zero or more arguments.

When a parameter declaration has a single `typedef` name in parentheses, it is not interpreted as redundant parentheses around the identifier for a declarator. Instead, it is interpreted as an abstract declarator that specifies a function with a single parameter.

If a parameter declaration contains a storage-class specifier in the declaration specifiers, then that storage-class specifier is ignored, except in the case in which the parameter declaration is one of the members of the parameter type list.

`register` is the only storage-class specifier that a parameter declaration can contain.

A function declarator may not return a function type or an array type.

Compatible Function Types

Two functions are compatible if the following are true:

- They both specify compatible return types.
- If both function types are prototyped (that is, if the parameter type lists are present in both), then they must have the same number of parameters and the same use of the ellipsis terminator. Corresponding parameters must have compatible types.

  However, if both function types are "old style," then the parameter types are not compared.
- If one function type is prototyped (that is, it contains a parameter list) and the other is old-style declaration that is not part of a definition (that is, it contains an empty identifier list), then there may be no ellipsis terminator in the parameter list, and the type of each parameter must be compatible with the resulting type after the execution of the default argument promotions.

- If one function type is prototyped (that is, it contains parameter type lists) and the other is old-style definition that contains an identifier list which may be empty, then they each have the same number of parameters. Also, the type of each prototyped parameter is compatible with the resulting type from the execution of the default argument promotions to the type of the corresponding identifier.

If a parameter is declared with qualified type, then its type for these comparisons is the unqualified version of its declared type.

In general, it is not advisable to mix old style function definitions and prototype function declarations of the same function. The following is an example of the old-style function definition followed by a prototype:

```
int f (s) /* old-style definition */
char s;
{

}


int f (char); /* incompatible prototyped
                 declaration */
```

These two types are not compatible because the parameter of the old-style function definition will result, after integral promotion, in type `int` instead of `char` in the second example.

Function Declarator Examples

In the following declaration

```
int f1(void), *f2(char), (*f3)(float);
```

- `f1` is declared as a function with no parameters returning `int`
- `f2` is declared as a function with one parameter of type `char` returning a pointer to an `int`
- a pointer `pfi` to a function with one parameter of type `float` returning an `int`

The extra parentheses are necessary in the declarator `(*f3)()` to indicate that indirection through a pointer to a function yields a function designator. That function designator is then used to call the function, which returns an `int`.

A declaration occurring outside of a function has identifiers with file scope and external linkage. A declaration in the form of `f1` or `f2` occurring inside a function has block scope and either internal or external linkage (depending on what file scope declarations for these identifiers are visible), and a declaration of the form of `f3` has block scope and no linkage.

In the following declaration, an array `apfi` is declared of four pointers to functions returning `int`.

```
int (*arr[4])(char *p1, char *p2);
```

Each of the functions returning `int` has two parameters that are pointers to `char`. (The identifiers `p1` and `p2` are declared only for clarification; those identifiers are out of the scope at the end of the declaration of `arr`.)

The following declaration declares a function `func` that returns a pointer to a function returning an `int`.

```
int (*func(int (*) (long), char)) (short, ...);
```

The function `func` has two parameters:

- a pointer to a function returning an `int`, which has one parameter of type `long`
- a `char`

`func` returns a pointer that points to a function with one `short` parameter. That function accepts a non-negative number of additional arguments of any type.

Type Names

A type name is used to specify a type without declaring an identifier.

A type name has the same syntax as either a function declaration or an object of that type without the identifier.

If a type name contains empty parentheses, the compiler will not treat them as unnecessary parentheses, surrounding an absent identifier, but instead as a function that does not specify parameters.

Type Name Syntax

type-name:
specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
pointer
pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
(*abstract-declarator*)
direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
direct-abstract-declarator$_{opt}$ (*parameter-type-list*$_{opt}$)

Type Name Examples

The following table illustrates the representation of each type.

TYPE	REPRESENTATION
`int`	`int`
pointer to `int`	`int *`
array of four pointers to `int`	`int *[4]`
pointer to an array of four `int`s	`int (*) [4]`
function with no parameter specifications returning a pointer `int`	`int *()`
pointer to function with no parameters returning `int`	`int (*) (void)`
const-pointer to array of pointer to function returning `int`	`int (*(*const)[]) ()`

The following example,

```
unsigned int (*const []) (char, ...)
```

declares an incomplete array of constant pointers to functions, each with one parameter that has type `char` and an unspecified number of other parameters, returning an `unsigned int`.

Type Definitions

If a declaration has `typedef` as its storage-class specifier, each declarator in that declaration defines an identifier to be a `typedef` name. That `typedef` name provides an alias or synonym of the specified type rather than a new type (see the discussion earlier in this chapter). `typedef`s may be useful to create synonyms for complicated types.

For example, in the following declaration:

```
typedef T type_ident;
type_ident D;
```

- the `typedef` name is `type_ident`
- `type_ident` has a type specified by the specifiers in T
- the identifier in D has the type *"derived-declarator-type-list* T"
- D specifies the *derived-declarator-type-list*

A `typedef` name shares the same name space as other ordinary identifiers. If the identifier is to be redeclared in an inner scope or is to be declared as a member of a structure or union in the same or an inner scope, the type specifiers must be given in the inner declaration.

`typedef` Declaration Syntax

typedef-name:
    identifier

typedef Declaration Examples

The following example declares three typedef names:

```
typedef int TIME, TIMEF();
typedef struct { int hour, sec; } time;
```

Once the typedef name is declared, the following constructions are possible:

```
TIME t;
extern TIMEF *tpfi;
time t1;
time t2, *tp;
```

In the preceding example,

- int is the type of t
- "pointer to function with no parameter specification returning int" is the type of tpfi
- the type of t1 is the specified structure
- the type of t2 is the specified structure
- tp is a pointer to the specified structure
- the type of t is compatible with any other object with type int

Assume the following declarations:

```
typedef struct s1 { char a; } struct1, *structp1;
typedef struct s2 { char a; } struct2, *structp2;
```

Then,

- The type struct1 and the type pointed to by structp1 are compatible.
- The type struct1 is compatible with type struct s1.
- The type struct1 is not compatible with any of the following types:

- — `struct s2, struct2`
- — the type pointed to by `structp2`
- — the type `int`

In the following constructions:

```
typedef signed int x;
typedef int ordinal;
struct tag {
    unsigned x:4;
    const x:5;
    ordinal y:5;
};
```

the following are true:

- A `typedef` name `x` is declared with type `signed int`.
- A `typedef` name `ordinal` is declared with type `int`.
- A structure is declared with three bit-field members.
  - — One of the bit-field members is named `x` and may contain values in the range [0,15].
  - — Another of the bit-field members is an unnamed const-qualified bit-field and could contain values in at least the range [−15,+15] (if it could be accessed).
  - — The third of the bit-field members is named `y` and may contain values in either the range [0,31] or [−15,+15], depending on whether a plain `int` bit-field is treated by the implementation as `unsigned` or `signed`; LPI-C will treat it as `unsigned`.

One difference between the first two of these bit-field members is that `unsigned` is a type specifier but `const` is a type qualifier. The `unsigned` type specifier forces `x` to be the name of a structure member. The `const` bit-field member modifies `x`, which is still in scope as a `typedef` name.

If the preceding declaration were followed by the following declaration in an inner scope:

```
x f(x (x));
long x;
```

then the function `f` would be declared with the type "function returning `signed int` with one unnamed parameter with type pointer to function returning `signed int` with one unnamed parameter with type `signed int`." That is:

```
signed int f(signed int (*) (signed int));
```

The identifier `x` would be declared with type `long`.

`typedef` names can be used to make code more readable. In the following example, all of the declarations of the `signal` function are of exactly the same type.

```
typedef void f(int);
typedef void (*pf) (int);

void (*signal(int, void (*) (int))) (int);
f *signal(int, f *);
pf signal(int,pf);
```

The following illustrates that type specifiers may not be used in conjunction with `typedef` names.

```
typedef    short int small;
unsigned   small object;      /* INVALID */
typedef    unsigned small;    /* INVALID */
```

The following illustrates some valid and invalid uses of type qualifiers to qualify `typedef` names.

```
typedef const int   CI; /* const-int */
typedef int        *PI; /* ptr-to-int */
typedef const CPI *PCI; /* ptr-to-const-int */
const CI             a; /* WRONG; dupl. type */
const volatile PI    b; /* const-volatile-ptr-
                           to-int */
const volatile PCI   c; /* const-volatile-ptr-
                           to-const-int */
const CPI            d; /* WRONG; dupl. type */
```

Initialization

Initialization provides the initial value that will be stored in a static or automatic duration object, prior to any subsequent modification by the program.

An object with automatic storage duration has indeterminate value, if it is not initialized explicitly.

An initializer for an object with static storage duration must be a constant expression. The entity being initialized must be an object type or be an array of unknown size.

A block scope declaration with internal or external linkage may not have an initializer.

All of the values in the initializer list for a structure or union object must be constant expressions.

Aggregate Initialization

The initializer for an object with aggregate type is a comma-separated list of initializers, enclosed in braces, for the aggregate members. Those aggregate members are presented in increasing order of subscript or member.

A character string literal, with or without braces, can be used to initialize an array of character type.

Each element of the array is initialized by each subsequent character of the character string literal, including the terminating null character, if the size of the array is not specified, or if there is enough room (as indicated by the specified array dimensions).

A wide string literal, with or without braces, can be used to initialize an array that has an element type compatible with `char_t`. Each member of the array may be initialized by each subsequent code of the wide string literal, including the terminating zero-valued code if the size of the array is unknown or if there is enough room.

If an initializer for a union object is enclosed by braces, it initializes the member that is present first in the declaration list of the union type. All unnamed structure or union members are ignored during initialization.

An initializer for a structure or union object with automatic storage duration is one of the following:

- an initializer list (as described in the following section)
- a single expression with compatible structure or union type

If the initializer contains a single expression, then the object will have as its initial value the same value as that of the expression.

Aggregate or Union Initializers

These rules apply recursively to any subaggregates or unions within the aggregate, or to an aggregate or union that is the first member of a union.

Brace-enclosed initializers of a subaggregate or contained union will initialize the members of the subaggregate or the member of the contained union that appears first.

Alternatively, the exact number of initializers from the list will be used to initialize the members of the first subaggregate or the member of the contained union that appears first. (If there are any unused initializers, they will be later used to initialize the next aggregate member that is a part of this subaggregate or contained union.)

If there are more members of an aggregate than there are initializers in a brace-enclosed list, then the unused members of the aggregate are initialized as though they were objects with static storage duration.

The size of an array of unknown size is determined by the number of initializers required by the array element. The array no longer has incomplete type when the end of its initializer list is reached.

Initialization of an object with static storage duration, if not achieved explicitly, is achieved implicitly as if each member with arithmetic type had the value 0 and each member with pointer type had the value of a null pointer constant.

An initializer list cannot contain more initializers than there are objects to be initialized.

Initialization Syntax

initializer:
    assignment-expression
    { *initializer-list* }
    { *initializer-list* , }

initializer-list:
    initializer
    initializer-list , *initializer*

Initialization Examples

In the following declaration, no size is specified and there are three initializers. `a` is therefore defined and initialized as a one-dimensional array object with three elements.

```
int a[] = { 2, 3, 5 };
```

The following example illustrates a definition with an initialization that is fully bracketed.

The first line (2, 3, 5) initializes the first column of the array object `b[0]`, that is, `b[0][0]`, `b[0][1]`, and `b[0][2]`.

The second line initializes `b[1]`.

The third line initializes `b[2]`. (Because the initializer ends early, zeros are used to initialize `b[3]`.)

```
long b[4][3] = {
    { 2, 3, 5 },
    { 8, 13, 21 },
    { 34, 55, 89 },
};
```

The following example illustrates another way of achieving the same effect.

```
long b[4][3] = {
    2, 3, 5, 8, 13, 21, 34, 55, 89
};
```

Because a left brace does not appear at the beginning of the initializer for `b[0]`, three items from the list are used for initialization. Similarly, the next three items are used for `b[1]`, and then the next three items are used for `b[2]`.

The following example initializes the first column of `c` as the declaration specifies. The remaining columns are initialized with zeros.

```
long c[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

The following example is a definition that contains an initialization that is not consistently bracketed.

This example defines an array in which all the elements are zero with the exception of two element structures: `s[0].x[0]` is 1 and `s[1].x[0]` is 2.

```
struct { int x[3], y; } s[] = { { 1 }, 2 };
```

The following declaration contains an initialization that is consistently but incompletely bracketed.

```
short m[4][3][2] = {
    { 2 },
    { 3, 5 },
    { 8, 13, 21 }
};
```

The preceding declaration defines a three-dimensional array object as follows:

- `m[0][0][0]` is 2
- `m[1][0][0]` is 3
- `m[1][0][1]` is 5
- 8, 13, and 21 initialize `m[2][0][0]`, `m[2][0][1]`, and `m[2][1][0]`, respectively
- all remaining intializers are zero

Because a left brace does not begin the initializer for `m[0][0]`, as many as six items may be used from the current list. However, because there is only one item in the list, the remaining five elements have their value initialized with zero.

Similarly, because a left brace does not begin the initializers for `m[1][0]` and `m[2][0]`, each can use as many as six items; thus each aggregate can initialize its two-dimensional subaggregates.

An error message would appear if there were more than six items in any of these lists.

The following two examples illustrate how the same initialization result could be achieved.

```
short m[4][3][2] = {
    2, 0, 0, 0, 0, 0,
    3, 5, 0, 0, 0, 0,
    8, 13, 21
};
```

The following example illustrates a fully-bracketed form that achieves the same initialization result:

```
short m[4][3][2] = {
    {
        { 2 },
    },
    {
        { 3, 5 },
    },
    {
        { 8, 13 },
        { 21 },
    }
    };
```

(Either the fully-bracketed or the minimally-bracketed initialization forms will generally cause less confusion.)

In the following example, the declaration defines "plain" `char` array objects `q` and `r`; character string literals have initialized the elements of those array objects.

```
char q[] = "xyz", r[3] = "xyz";
```

The result of the preceding declaration is the same as is achieved by the following declaration, in which the contents of the arrays are modifiable.

```
char q[] = { 'x', 'y', 'z', '0' },
     r[] = { 'x', 'y', 'z' };
```

Alternatively, the following example shows that the declaration defines `p` with type "pointer to `char`." `p` is initialized to point to an object with type "array of `char`" that has a length of 4 and has elements that are initialized with a character string literal. (Any attempt to use `p` to modify the contents of the array will result in undefined behavior.)

```
char *p = "xyz";
```

Chapter 8: Conversions

Chapter 8: Conversions

Overview

Conversion refers to the implicit or explicit changing of values from one type to another. Implicit conversion occurs when a operator converts operand values automatically. Explicit conversion results from a cast operation.

This chapter describes the results of such conversions.

The value and the representation do not change as the result of a conversion of a value to a compatible type.

Arithmetic Operands

This section describes arithmetic operands, which include characters and integers, as well as floating and integral types.

Integral Promotions

Integral promotions convert operands of the types listed below to either an `int` or `unsigned int` value when used in any expression where an `int` or `unsigned int` may be used:

- signed or unsigned `char`
- signed or unsigned `short int`
- signed or unsigned `int` bit-field
- object with enumeration type

The value is converted to an `int`, if all values of the original type can be so represented. Otherwise, the value is converted to an `unsigned int`.

These promotions occur as part of the "usual arithmetic conversions" in certain argument expressions to the operands of the unary +, −, and (tilde) ~ operators and to both operands of the shift operators. The preserved value of an integral promotion also includes the sign. Whether "plain" `char` is treated as signed or unsigned is implementation-defined; note that LPI-C always treats a "plain" `char` as signed. Integral promotions do not change other arithmetic types.

Note
ANSI C uses what is referred to as "value-preserving" rules for the integral promotions. Some older C compiler implementations used "unsigned-preserving" rules. Refer to Chapter 12 for details.

Signed and Unsigned Integers

The value of an integral type that is converted to another integral type is unchanged, if the new type can represent its value.

A non-negative value of a signed integer that is converted to an unsigned integer with equal or greater size is unchanged. However, if the value of the signed integer is negative, the following occurs:

1. The signed integer is promoted to the signed integer corresponding to the unsigned integer.
2. The value is converted to unsigned. This conversion occurs using the following formula:

   $$i + (n+1)$$

   where, i is the value and n is the largest number that can be represented in the unsigned integer type.

   Note
   In a two's-complement representation, this is equivalent to filling the high-order bits with copies of the sign bit.

The result of a conversion of a value with integral type to an unsigned integer with smaller size is represented by the nonnegative remainder in the following formula:

$$i/(n+1)$$

where, *i* is the value and *n* is the the largest unsigned number that can be represented in the type with smaller size.

The results of the following conversions and conditions are implementation-defined if the value cannot be represented. (See Appendix C, item 5.ii, in the *LPI-C User's Guide.*)

- a value with integral type is changed to a smaller-sized signed integer
- an unsigned integer is converted to its corresponding signed integer

Floating and Integral Types

The fractional part of a floating type value is discarded when the value is converted to an integral type. The behavior is undefined if the remaining value of the integral part of the floating type cannot be represented by the integral type.

Note that the range of portable floating values used in floating type conversions to unsigned type is 0 to the maximum value of that unsigned type.

For integral to floating type conversions, if the value being converted can be represented but not exactly, the result is implementation-defined to be either the nearest higher or nearest lower value. (See Appendix C, item 6.ii, in the *LPI-C User's Guide.*)

Float to Double Promotions

The following promotions result in unchanged values:

- `float` to `double`
- `float` to `long double`
- `double` to `long double`

The following conversions also result in unchanged values, except if the value being converted is outside the range of representable values, in which case the behavior is undefined, or if the value being converted can be represented but not exactly, in which case the result is implementation-defined to be either the nearest higher or nearest lower value (see Appendix C, item 6.iii, in the *LPI-C User's Guide*).

- `double` to `float`
- `long double` to `double`
- `long double` to `float`

Arithmetic Conversions

Many operators that use arithmetic type operands create conversions and yield result types similarly. The goal is to produce common types, including the result type. This conversion pattern is referred to as the "usual arithmetic conversions." The conversions are as follows and are applied in the given order.

If either operand is:	The other is converted to:
`long double`	`long double`
`double`	`double`
`float`	`float`

Otherwise, the integral promotions are performed on both operands and the following is applied to the resultant operands:

If either operand is:	The other is converted to:
`unsigned long int`	`unsigned long int`

Otherwise, if one operand is a `long int` and the other is an `unsigned int` and if a `long int` can represent all values of an `unsigned int`, the `unsigned int` is converted to `long int`. If not, both operands are converted to `unsigned long int`.

Otherwise,

If either operand is:	The other is converted to:
`long int`	`long int`
`unsigned int`	`unsigned int`

If none of the previous conditions are true, both operands have type `int`.

Other Operands

This section describes non-arithmetic operands.

lvalues and Function Designators

An lvalue designates an object. An lvalue expression has an object type or an incomplete type other than `void`. The term "lvalue" stems from the left operand of an assignment expression which must be a modifiable value.

An identifier of an object is a simple example of an lvalue. Another example is `*p`, where `p` is a unary expression that is a pointer to an object. `*p` is an lvalue that designates the object pointed to by `p`.

An lvalue specifies the particular type used to designate an object, when an object has a particular type. lvalues that do **not** have the following type are known as modifiable lvalues:

- an array type
- an incomplete type
- a const-qualified type
- if the lvalue is a structure or union, does not have any member that has a const-qualified type including, recursively, any member of all contained structures or unions

An lvalue that does not have array type is converted to the value stored in the designated object and is no longer an lvalue, except when it is the operand of one of the following:

- the `sizeof` operator
- the unary `&` operator
- the `++` operator
- the `--` operator
- the left operand of the `.` operator
- the left operand of an assignment operator

The value of an lvalue with qualified type has the unqualified version of the type of the lvalue. If the lvalue does not have qualified type, the value has the type of the lvalue. The behavior is undefined if the lvalue has an incomplete type and does not have array type.

An lvalue that has type "array of type" is converted to an expression that has type "pointer to type." The converted expression points to the initial element of the array object and is not an lvalue, except when it is one of the following:

- the operand of the `sizeof` operator
- the operand of the unary `&` operator
- a character string literal used to initialize an array of character type
- a wide string literal used to initialize an array with element type compatible with `wchar_t`

An expression with function type is referred to as a function designator. A function designator with type "function returning type" is always converted to an expression that has type "pointer to function returning type" except when the following is true:

- it is the operand of the `sizeof` operator
- it is the operand of the unary `&` operator

`void`

A `void` expression refers to an expression that has type `void`. The value of a `void` expression is by definition nonexistent, and implicit or explicit conversions (except to `void`) are not applied to such an expression. The value or designator of an expression of any other type that occurs in a context where a `void` expression is required is discarded.

`void` expressions are evaluated for possible side effects.

Pointers

A pointer to `void` may be converted to a pointer to any incomplete or object type or from a pointer to any incomplete or object type and back again. The value of the result of such a conversion is equal to the value of the original pointer.

For any qualifier type, a pointer to a non-qualified type may be converted to a pointer to the qualified version of type. Again, the value of the result of such a conversion is equal to the value of the original pointer.

Null Pointer

A null pointer constant is the same as an integral constant expression with the value 0, or such an expression cast to type `void *`. The constant is converted to a pointer of the type it is assigned to or compared for equality to. A null pointer (as it is referred to) will always compare unequal to a pointer to any object or function.

Two null pointers compare equal even though they may have been converted through different sequences of casts to pointer types.

Chapter 9: Expressions

Chapter 9: Expressions (Cont.)

Chapter 9: Expressions

Overview

An expression is a sequence of operators and operands that performs one or more of the following:

- computation of a value
- designation of an object or a function
- generation of a side effect

Side Effects and Sequence Points

Evaluation of an expression may produce side effects, which are changes in the state of the execution environment. Side effects are any of the following:

- accessing a volatile object
- modifying an object
- modifying a file
- calling a function that does any of these operations

Sequence points are those specified points in the execution sequence where all side effects of previous evaluations are complete and the side effects of any subsequent evaluations have not yet taken place.

Places where sequence points exist include:

- immediately after all the arguments have been evaluated and before a function call
- after the evaluation of the first operand of a logical AND (&&) expression
- after the evaluation of the first operand of a logical OR (||) expression
- after the evaluation of the first operand of a conditional (? :) expression

- after the evaluation of the first operand of a comma (,) expression
- at the end of a full expression (that is, an expression that is not part of another expression), including the following:
  - an initializer
  - an expression statement
  - the expression controlling an `if` or `switch` statement
  - the expression controlling a `do` or `while` statement
  - each of the three expressions in a `for` statement
  - the expression in a `return` statement

The stored value of an object may be modified only once by the evaluation of an expression between the previous and next sequence point. In addition, the previous value is only accessed to determine a new value that will be stored.

Thus, the following statement expression is undefined:

```
i = ++i + 1;
```

The order of evaluation of subexpressions is unspecified, as is the order in which side effects take place, except as indicated by the syntax or when the following operators are used, each of which is discussed later in this chapter:

- the function-call operator (())
- the logical AND (&&) operator
- the logical OR (||) operator
- the conditional (?:) operator
- and the comma (,) operator

An expression in the abstract machine is evaluated as specified by the semantics. The implementation may eliminate the evaluation of a subexpression if it can determine that the value or side effects of that subexpression will not be used.

If a signal interrupts the processing between two sequence points, there may be objects whose values are currently being modified. In this case, the values of objects can only be relied upon as of the previous sequence point.

Each time the program enters a block, any object with automatic storage duration will have its last-stored value retained until the end of the execution of the block, as well as while the block is suspended by a call of a function or receipt of a signal.

The rules regarding sequence points guarantee the following:

- Volatile objects are stable at sequence points. That is, at sequence points, all previous evaluations have been completed and all subsequent evaluations have not yet occurred.
- All data written into files at program termination are the same as if that data had been produced by execution of the program according to the abstract semantics.

An evaluation is only partly determined by its grouping. For example, in the following program fragment:

```
#include <stdio.h>
int sum;char *p;
/*...*/
sum = sum * 10 - '0' + (*p++ = getchar());
```

the expression statement is grouped as if it were written as follows:

```
sum = (((sum * 10) - '0') + ((*p++)) =
     (getchar())));
```

However, the call to `getchar` can occur at any point before its returned value is needed.

Also, the actual increment of `p` can occur at any time between the previous sequence point and the next sequence point.

Evaluation of an Expression

When an expression is evaluated, the precedence of operators is specified by the syntax. Table 9-1 outlines the precedence of operators in the evaluation of an expression in C.

The order in which the operators are described in this chapter also reflects their order of precedence; the operator with highest precedence is described first.

Exceptions to the rules of precedence are cast expressions as operands of unary operator, and operands which are contained between any of the following pairs of operators:

- grouping parentheses ( )
- subscripting brackets [ ]
- function-call parentheses ( )
- the conditional operator ? :

The following operators have operands with integral types:

- unary operator (~)
- binary (or bitwise) operators (<<, >>, &, ^)
- vertical bar (|)

Values returned by the operators in the preceding list will depend on the internal representations of integers. Those values, therefore, will have implementation-defined aspects for signed types.

The behavior of the evaluation of an expression is undefined if the resulting value is not mathematically defined or not representable for its type.

Precedence and Associativity of Operators

The following table shows the C operators in descending order of precedence by operator group. Operator groups are delineated by horizontal lines. Operators within each group have equal precedence.

TABLE 9-1 Precedence and Associativity of Operators

OPERATOR	DESCRIPTION	ASSOCIATIVITY
()	Function call	Left to right
[]	Reference to array element	
.	Reference to structure member	

OPERATOR	DESCRIPTION	ASSOCIATIVITY
`->`	Structure or union member reference	
`+`	Unary plus	Right to left
`-`	Unary minus	Right to left
`++`	Increment (pre/post)	
`--`	Decrement (pre/post)	
`!`	Logical negation	
`~`	Bitwise complement	
`*`	Pointer indirection	
`&`	Address	
`sizeof`	Object size in bytes	
(*type*)	Type coercion (cast)	
`*`	Multiply	Left to right
`/`	Divide	
`%`	Remainder	
`+`	Add	Left to right
`-`	Subtract	
`<<`	Left shift	Left to right
`>>`	Right shift	
`<`	Less than	Left to right
`<=`	Less than or equal to	
`>`	Greater than	
`>=`	Greater than or equal to	
`==`	Equality	Left to right
`!=`	Inequality	
`&`	Bitwise AND	Left to right
`^`	Bitwise XOR	Left to right
`|`	Bitwise OR	Left to right
`&&`	Logical AND	Left to right
`||`	Logical OR	Left to right
`?:`	Conditional expression	Right to left
`=` `*=` `/=` `%=` `+=` `-=` `&=` `^=` `|=` `<<=` `>>=`	Assignment operators	Right to left
`,`	Comma	Left to right

Primary Expressions

A primary expression can be one of the following:

- an identifier that is an lvalue (that is, an identifier that has been declared as designating an object)
- an identifier that is a function (that is, a function designator)
- a constant (see the section "Constants" in Chapter 3)
- a string literal (see the section "String Literals" in Chapter 3)
- a parenthesized expression (the type and value of a parenthesized expression are the same as the type and value of an unparenthesized expression)

Primary Expression Syntax

primary-expression:
identifier
constant
string-literal
(*expression*)

Postfix Expressions

This section describes postfix operators, including the following:

- array subscripting
- function calls
- structure and union members
- postfix increment and decrement operators

Postfix Expression Syntax

postfix-expression:
primary-expression
postfix-expression [*expression*]
postfix-expression (*argument-expression-list*$_{opt}$)
postfix-expression . *identifier*
postfix-expression -> *identifier*

postfix-expression ++
postfix-expression --

argument-expression-list:
assignment-expression
argument-expression-list , *assignment-expression*

Array Subscripting

A subscripted element of an array is represented by a postfix expression that is followed by an expression in square brackets [ ].

The subscript operator [ ] is defined so that the following expression:

```
E1 [E2]
```

is equivalent to the following expression:

```
(*(E1+(E2)))
```

If E1 is an array object, or a pointer to the first member of an array object, and E2 is an integer, then E1 [E2] will designate the E2-th element of E1 starting at zero.

An element of a multi-dimensional array object is designated by successive subscript operators. For example, if E is an n-dimensional array ($n \geq 2$) that has the following dimensions:

$$i \times j \times \ldots \times k$$

and if E is not an lvalue, it is converted to a pointer to an $(n-1)$-dimensional array with the following dimensions:

$$j \times \ldots \times k$$

In this example, if the unary * operator is then applied to this pointer explicitly (or if it is applied implicitly as a result of subscripting), the result is the pointed-to $(n-1)$-dimensional array. If that array is not an lvalue, it is itself converted into a pointer. Arrays are stored in row-major order (the last subscript varies fastest).

The types of the operands in an array subscript operation must be as follows:

- one must be an expression of type "pointer to object *type*"
- the other must be an expression of integral type

The result will have type "*type*."

Array Object Example

The following declaration defines an array object:

```
int x[3][5];
```

where `x` is a 3×5 array of `int`s. That is, `x` is an array of three element objects and each element object is an array of five `int`s.

The following expression:

```
x[i]
```

is equivalent to the following:

```
(*(x+(i)))
```

and `x` is converted to a pointer to the initial array of five `int`s.

Then `i` is adjusted according to the type of `x`. This adjustment involves multiplying `i` by the size of the pointed-to object (that is, an array of five `int` objects).

The results of this multiplication are added and indirection is applied to yield an array of five `int`s.

When that array is used in the expression `x[i][j]`, that expression is in turn converted to a pointer to the first of the `int`s, and `x[i][j]` yields an `int`.

Function Calls

A function call is represented by either of the following:

- A postfix expression that specifies the called function.

- Parentheses following the postfix expression; those parentheses can be empty or can contain a comma-separated list of expressions. The list of expressions specifies the arguments to the function. Each of the arguments may be of any object type.

If the postfix expression contains only an identifier, and if there is no current declaration for that identifier, then the identifier is implicitly declared exactly as if it were declared in the innermost block containing the function call as follows:

```
extern int identifier();
```

Before a function is called, all arguments are evaluated. Each parameter is assigned the value of its corresponding argument.

The values of the parameters may be changed by the function. These changes will not affect the values of the arguments; this is known as "pass-by-value." If, on the other hand, a pointer to an object is passed as an argument, it is possible for the called function to change the value of the object to which the argument points.

Function Prototypes

A function prototype is a function declaration which declares its parameter types.

If a called function has been declared without a prototype, then it is referred to as an old-style function declaration; in this case, the integral promotions are performed on each argument. Arguments with type `float` are promoted to `double`; these are referred to as the default argument promotions.

The behavior is undefined in the following circumstances:

- when the number of arguments and the number of parameters are not the same
- when the function is defined with a type that does not include a prototype and the types of the arguments after promotion are not compatible with the types of the parameters after promotion

- when the function is defined with a type that includes a prototype and the types of the arguments after promotion are not compatible with the types of the parameters
- when the function is defined with a type that includes a prototype and the prototype ends with an ellipsis (...)

In an expression that calls a function that was previously declared with a prototype,

- The number of arguments must be the same as the number of parameters.
- The type of each argument must be compatible with the unqualified version of the type of the corresponding parameter.
- Arguments are implicitly converted to the types of the parameters to which they correspond; that is, they are converted as if by assignment.
- The presence of a trailing ellipsis in a prototyped declaration causes argument type conversion to be terminated after the last declared parameter. Trailing arguments are subjected to the default argument promotions.

The behavior is undefined if the return type of the function definition is not compatible with the return type of the expression pointed to by the expression that calls the function.

In an old-style function definition (that is, one that does not include a function prototype declarator), the number and types of arguments are not compared with the number and types of the corresponding parameters.

All arguments are completely evaluated before the function call. However, the order of evaluation is unspecified for the function designator, the arguments, and subexpressions within the arguments.

A recursive function call can be executed as the result of any chain of other functions.

The expression that calls a function (usually resulting from the conversion of a function designator) will have one of the following types:

- pointer to function returning `void`
- pointer to function returning some object type that is not array type

Function Expression Example

In this example, the functions `f1`, `f2`, `f3`, and `f4` may be called in any order:

```
(*pf[f1()]) (f2(), f3() + f4())
```

Before the function pointed to by `pf[f1()]` is entered, all side effects are completed.

Structure and Union Members

A structure or union object is designated by one of the following forms:

- A postfix expression that is followed by a dot (`.`) and an identifier. Its value is that of the named member and it is an lvalue if the first expression is also an lvalue.

  If the first expression of the structure or union object has qualified type, then the result will have the type of the designated member that is qualified in the same way.

  The first operand of the `.` operator must have a structure or union type that is either qualified or unqualified; the second operand must name a member of that type.

- A postfix expression that is followed by an arrow `->` and an identifier, in which the value (an lvalue) is the same as the value of the named member of the object pointed to by the first expression.

  For example, if `&E` is a valid pointer expression, in which `&` represents the "address-of" operator and generates a pointer to the operand, then the expression `(&E)->MOS` is the same as `E.MOS`.

If the first expression of the structure or union object is a pointer to a qualified type, then the result will have the type of the designated member that is qualified in the same way.

The first operand of the `->` operator must have a type that is a pointer to a structure or a union, either qualified or unqualified; the second operand names a member of the type pointed to by the first operand.

The behavior is implementation-defined if the value of a member of a union object is accessed after the value has been stored in a different member of the object. For example, byte ordering of different types may be dependent upon externally generated data, but otherwise would be consistent within self-contained programs.

An exception to this rule is made to simplify the use of unions: a union object can inspect the common initial part of the union structures that share a common initial sequence.

To share a common initial sequence, two or more structures must have a sequence of one or more initial corresponding members with compatible types or initial corresponding bit-fields with the same widths.

Structure and Union Examples

- `f().x` is a valid postfix expression (but is not an lvalue) if:

  `f` is a function returning a structure or union

  `x` is a member of that structure or union

- The following is a valid fragment:

```
union {
     struct {
          int     key;
     } n;
struct {
          int     keynum;
          char    name;
     } nf;
} u;
```

```
/*...*/
u.nf.type = 1;
u.nf.name = "Grendel";
/*...*/
if (u.n.key == 1)
     /*...*/ search(u.nf.name) /*...*/
```

Postfix Increment and Decrement Operators

The postfix ++ operator will add the value 1 of the appropriate type to the operand after the value has been obtained. That operand must be an lvalue that can be modified and must be of qualified or unqualified scalar type.

The postfix -- operator will subtract the value 1 of the appropriate type from the operand after the value has been obtained.

The side effect of postfix incrementing or decrementing occurs between the prior and the subsequent sequence points.

Unary Operators

The syntax of unary operators follows:

unary-expression:
postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator cast-expression
sizeof *unary-expression*
sizeof **(** *type-name* **)**

unary-operator: one of
& ***** **+** **-** **~** **!**

Prefix Increment and Decrement Operators

The prefix ++ operator adds the value 1 of the appropriate type to the operand before the value has been obtained. That operand must be an lvalue that can be modified and must have qualified or unqualified scalar type.

The incremented value will become the new value of the operand. Thus, the expression `++E` is equivalent to `(E+=1)`.

Likewise, the prefix `--` operator subtracts the value 1 of the appropriate type from the operand before the value has been obtained. Thus, the expression `--E` is equivalent to `(E-=1)`.

Address and Indirection Operators

The unary **&** (address-of) operator yields a pointer to the object or function that the operand designates. An operand with type "*type*," will yield a result with type "pointer to *type*."

The operand of the unary * operator must have pointer type and yields the following:

- a function designator, if the operand points to a function
- an lvalue that designates an object, if it points to an object
- a result with type "*type*," if the operand has type "pointer to *type*"

The behavior of the unary * operator is undefined if the value assigned to the pointer is invalid.

The following values are invalid for de-referencing a pointer by the unary * operator:

- a null pointer
- an address that is not properly aligned for the type of object pointed to
- the address of an object with automatic storage duration, if the execution has terminated of the block with which the object is associated

The operand of the unary & operator must be one of the following:

- a function designator
- an lvalue that designates an object that is neither a bit-field nor is declared with the `register` storage-class specifier.

If E is one of the following:

- a function designator
- an lvalue that is a valid operand of the unary & operator

then *&E is equivalent to E and is a function designator or an lvalue, respectively.

Unary Arithmetic Operators

Each unary arithmetic operator modifies its operand as shown in the following table.

TABLE 9-2 Unary Arithmetic Operators

UNARY OPERATOR	RESULT
+	the value of its operand after the integral promotions have been performed
-	the negative of its operand after the integral promotions have been performed
~	bitwise complement of its operand after the integral promotions have been performed
!	if the value of the operand is 0, then the result is 1; otherwise, the result is 0 Thus, the expression !E is equivalent to (E==0)

If E is promoted to type `unsigned long`, then ~E is the same as `(ULONG_MAX - E)`.

If E is promoted to type `unsigned int`, then E is equivalent to `(UINT_MAX - E)`. (`ULONG_MAX` and `UINT_MAX` are defined in the header `<limits.h>`.)

Operands associated with the unary operators must have specific types, as shown in the following table.

UNARY OPERATOR	OPERAND TYPE
+	arithmetic
–	arithmetic
~	integral
!	scalar

The `sizeof` Operator

The **`sizeof`** operator is used to yield the size of an operand, with the result (in bytes) as an integer constant.

The **`sizeof`** operator may be applied to an operand that is one of the following:

- an expression that does not have function type or incomplete type
- the parenthesized name of a type that is not function or incomplete type

The **`sizeof`** operator may not be applied to an lvalue denoting a bit-field object.

The type of the operand is evaluated, not the operand itself, and it is the operand type that will determine the size.

The result is 1 if the evaluated operand has one of the following types (or a qualified version of one of these types), including the following:

- **`char`**
- **`unsigned char`**
- **`signed char`**

If the operand has array type, then the result is the total number of bytes in the array. If the operand is a parameter with array or function type, then the result is the size of the pointer obtained after conversion.

If an operand has structure or union type, the result of the application of the `sizeof` operator is the total number of bytes in that object. This includes all internal and trailing padding for alignment purposes, which is implementation-defined. (See Appendix C, item 7.i, in the *LPI-C User's Guide.*) The type of that result is `size_t`, which is defined in the header `<stddef.h>`.

The `sizeof` operator is typically used when computing the number of elements in an array, as in the following example:

```
sizeof (array) / sizeof (array[0])
```

Cast Operators

A cast is an expression preceded by a type name enclosed in parentheses. The value of that expression is converted or coerced to the named type. If a cast does not specify a conversion, then the type or value of the expression is not modified.

The type name must specify a qualified or unqualified scalar type, or a `void` type. Whether a cast is applied to a qualified or to an unqualified version of a type, the result will be the same. An lvalue cannot result from a cast.

The associated operand must have scalar type.

Cast Conversion of Pointers

Applying a cast to convert a pointer has certain implementation-defined and undefined characteristics, as follows:

- The result of a cast and the size of the integer required for the cast are implementation-defined when a pointer is converted to an integral type. (See Appendix C, item 7.ii, in the *LPI-C User's Guide.*)

  The behavior is undefined if there is insufficient space for the conversion.

- The result of a cast is implementation-defined when an arbitrary integer is converted to a pointer. (See Appendix C, item 7.ii, in the *LPI-C User's Guide.*)

Note that conversion of a pointer to an integer and vice versa should be consistent with the addressing architecture of the machine.

- The result of a cast is undefined when a pointer to an object or an incomplete type is converted to a pointer to a different object or incomplete type. The resulting pointer may, for example, be improperly aligned for the type that is pointed to.

  However, a pointer to an object can be converted to a pointer to another object that has the same alignment (or a less strict alignment), and then can be converted back again, and the result will have a value that is the same as the original pointer. The object with the least strict alignment is the object with character type.

  Likewise, a pointer to a function of one type may be converted to a pointer to a function of another type and then converted back to its original type; the result will have the same value as the original pointer. However, the behavior is undefined if a converted pointer is used to call a function with a type that is not compatible with the type of the called function.

Cast Syntax

cast-expression:
unary-expression
(*type-name*) *cast-expression*

Multiplicative Operators

Operands associated with the multiplicative operators must have specific types, as shown in the following table.

MULTIPLICATIVE OPERATOR	OPERAND TYPES
*	arithmetic
/	arithmetic
%	integral

When multiplicative operators are used, the usual arithmetic conversions are performed on the operands and the results are as presented in the following table.

OPERATOR	RESULT
*	the product of the operands
/	the quotient from the division of the first operand by the second
%	the remainder of the first operand modulo the second operand

For both the / operator and the % operator, the behavior is undefined if the second operand has a value of zero. The sign of the result of the % operator is implementation-defined. (See Appendix C, item 5.iii, in the *LPI-C User's Guide.*)

If the division of two positive integers yields an inexact result, the results are as follows:

- The result of the / operator is the largest integer that is less than the algebraic quotient.
- The result of the % operator is positive.

If the division of two negative integers yields an inexact result, the result of the / operator is implementation-defined (see Appendix C, item 5.v, in the *LPI-C User's Guide*) and is one of the following:

- the largest integer that is less than the algebraic quotient
- the smallest integer that is greater than the algebraic quotient

For example, the following expression is equivalent to `a`, unless the result of `a/b` is not representable:

```
(a/b)*b + a%b
```

Multiplicative Syntax

multiplicative-expression:
cast-expression
multiplicative-expression * *cast-expression*
multiplicative-expression / *cast-expression*
multiplicative-expression % *cast-expression*

Additive Operators

Operands associated with the additive operators must have specific types, as shown in the following table.

ADDITIVE OPERATOR	OPERAND TYPES
+	both operands must have arithmetic type or one operand must have integral type and one operand must be a pointer to an object type
-	both operands must have arithmetic type or both operands must be pointers to qualified or unqualified versions of compatible object types or the left operand must be a pointer to an object type and the right operand must have integral type

When additive operators are used, arithmetic conversions are performed on the operands and the results are as presented in the following table.

(Using the ++ operator to increment is equivalent to adding 1. Using the -- operator to decrement is equivalent to subtracting 1.)

OPERATOR	RESULT
+	the sum of the operands
-	the remainder after the second operand is subtracted from the first operand

Pointer Arithmetic

If an expression with integral type is added to or subtracted from a pointer, then the result will have the same type as the pointer operand.

If that pointer operand points to an element of an array object, and if the array object is large enough, then the result will point to an element that differs from the original element in such a way that the difference between the subscripts of the original and resulting array elements is equal to the integral expression.

For example, if the pointer expression `p` points to the *i*-th element of an array object, then the following expression points to the *i*+n-th element of the array object (if that object exists):

```
(p)+n  (or  n+(p))
```

Likewise, the following expression points to the *i*—n-th element of the array object, if that object exists:

```
(p)-n
```

Also, when the expression `p` points to the last element of an array object, then the expression `(p)+1` points to one past the last element of the array object.

When the expression `q` points to an element that is one past the last element of the array object, then the expression `(q)-1` points to the last element of the array object.

If the result does not point to an element within the array, and it is dereferenced using the indirection operator, then the behavior is undefined.

Subtraction of one pointer from another, where both pointers point to elements of the same array object, results in the difference between the subscripts of the two array elements. The type of the result is `ptrdiff_t`, as defined in the `<stddef.h>` header.

For example, if the expression `p` points to the *i*-th element of an array object and `q` points to the *j*-th element of an array object and if the resulting value fits in an object that has type `ptrdiff_t`, then the following expression has the value *i-j*:

```
(p)-(q)
```

The behavior is undefined if both pointers do not point to elements within the same array.

Additive Syntax

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression − *multiplicative-expression*

Bitwise Shift Operators

After the integral promotions are performed on both operands, the result has a type that is the same as the type of the promoted left operand. The behavior is undefined if the right operand has a negative value or the right operand has a value that is greater than or equal to the width (measured in bits) of the promoted left operand.

The expression `E1 << E2` yields the following results:

`E1` left-shifted `E2` bit positions zero-filling vacated bits

If `E1` has an unsigned type, then the value of the result is determined as follows:

- `E1` is multiplied by the quantity, 2 raised to the `E2` power
- if `E1` has type `unsigned long`, the result is reduced modulo `ULONG_MAX+1`

If E1 does not have type `unsigned long`, the result is reduced modulo `UINT_MAX+1`. (The constants `ULONG_MAX` and `UINT_MAX` are defined in the header `<limits.h>`.)

The expression `E1 >> E2` yields the following result:

- E1 right-shifted E2 bit positions

If E1 is either an `unsigned` type or a `signed` type and a negative value, the value of the result is determined by:

- the integral part of the quotient of E1 divided by the quantity, 2 raised to the E2 power

The result is implementation-defined if E1 has a signed type and a negative value. (See Appendix C, item 5.iv, in the *LPI-C User's Guide*.)

Bitwise Shift Syntax

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

Relational Operators

Relational operators compare two operands.

Operands associated with relational operators must be one of the following:

- two operands with arithmetic type
- two operands that are pointers to qualified or unqualified versions of object types that are compatible
- two operands that are pointers to qualified or unqualified versions of incomplete types that are compatible

Relational operators yield results with type `int` and compare equal to 1 if the specified relation is true and 0 if it is false.

Relational Pointers

The result of the relational comparison of two pointers will express the relative locations of the objects pointed to within the address space.

A pointer to an object that is not part of an array behaves in the same way as a pointer to the first element of an array that has a length of one and an element type which is the same type as the object.

When the objects pointed to are members of the same aggregate object, then the pointers to structure members that are declared later in the structure will compare higher than pointers to members declared earlier in the structure.

Pointers to array elements that have larger subscript values will compare higher than pointers to elements of the same array that have lower subscript values.

Pointers to members of the same union object will compare equal.

The result is undefined if the objects pointed to are not members of the same aggregate or union object.

Relational Syntax

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*

Equality Operators

Equality operators compare two operands and behave in much the same manner as relational operators. Equality operators, however, have lower precedence than relational operators.

Operands used by the equality operators are under the following constraints:

- both operands must have arithmetic type
- both operands must be pointers to qualified or unqualified versions of compatible types
- one operand must be a pointer to an object or incomplete type and the other must be a qualified or unqualified version of `void`
- one operand must be a pointer and the other must be a null pointer constant

(See the section "Relational Operators" for information concerning the appropriate types and values of associated operands.)

Equality Operators and Pointers

Two pointers compare equal if and only if one of the following conditions is true:

- they both point to the same object or incomplete types
- they both point to the same element, which is located one after the last element of the same array

A pointer to an object or incomplete type is converted to the type of the other operand if the other operand has type pointer to a qualified or unqualified version of `void`.

Equality Syntax

equality-expression:
relational-expression
equality-expression == relational-expression
equality-expression != relational-expression

Bitwise AND Operator

The bitwise AND of the operands, which must have integral type, is the result when the bitwise `&` operator is applied, after the usual arithmetic conversions are performed on the operands.

Each of the resulting bits is set if and only if the corresponding bit in the converted operand is set.

Bitwise AND Syntax

bitwise-AND-expression:
equality-expression
bitwise-AND-expression & *equality-expression*

Bitwise exclusive OR Operator

The bitwise exclusive OR (also called XOR) of the operands, which must have integral type, is the result of the ^ operator, after the usual arithmetic conversions have been performed on the operands.

Each of the resulting bits is set if and only if exactly one of the corresponding bits in the converted operands is set.

Bitwise exclusive OR Syntax

bitwise-XOR-expression:
bitwise-AND-expression
bitwise-XOR-expression ^ *bitwise-AND-expression*

Bitwise inclusive OR operator

The bitwise inclusive OR of the operands, which must have integral type, is the result of the | operator, after the usual arithmetic conversions have been performed on the operands.

Each of the resulting bits is set if and only if at least one of the corresponding bits in the converted operands is set.

Bitwise inclusive OR Syntax

bitwise-OR-expression:
bitwise-XOR-expression
bitwise-OR-expression | bitwise-OR-expression

Logical AND operator

The result of the `&&` operator is 1 and type `int` if neither of the operands, which must have scalar type, is equal to 0; otherwise, the result is 0. The resulting type is `int`.

The `&&` operator always evaluates from left to right, unlike the bitwise binary `&` operator. After the first operand is evaluated, there is a sequence point. The second operand is not evaluated if the first operand is equal to 0.

This guarantee of logical expression "short-circuit" can be extremely useful. For example, in the following statement

```
while (!feof(f) && ((c = getc(f)) == '\n')) ;
```

the function `getc` will not be called if the expression `!feof(f)` is false (that is, the expression `feof` is true), thereby ensuring that no attempt to read past the end-of-file is made.

Logical AND Syntax

logical-AND-expression:
bitwise-OR-expression
logical-AND-expression `&&` bitwise-OR-expression

Logical OR Operator

The result of the `||` operator is 1 and type `int` if either one or both of the operands, which must have scalar type, is not equal to 0; otherwise, the result is 0. The resulting type is `int`.

This guarantee of logical expression "short-circuit" can be extremely useful. For example, in the following statement:

```
while (feof(f) || ((c = getc(f)) == '0')) ;
```

the function `getc` will not be called if the expression `feof(f)` is true, thereby ensuring that no attempt to read past the end-of-file is made.

Logical OR Syntax

logical-OR-expression:
logical-AND-expression
logical-OR-expression `||` *logical-AND-expression*

Conditional Operator

The conditional operator is used to evaluate three operands.

The first of the three operands must have scalar type. For the second and third operands, one of the following must be true:

- both operands have arithmetic type
- both operands have compatible structure or union types
- both operands have `void` type
- both operands are pointers to qualified or unqualified versions of compatible types
- one operand is a pointer and the other is a null pointer constant
- one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`

The conditional expression evaluation proceeds as follows:

- the first operand is evaluated (there is a sequence point after its evaluation)
- if the first operand is not equal to 0, then the second operand is evaluated
- if the first operand is equal to 0, then the third operand is evaluated

The result is the value of the operand that is evaluated (either the second or third operand). The result is not an lvalue.

In cases where both the second and third operands have arithmetic type, the usual arithmetic conversions are performed to bring them to a common type. The result will have that type.

In cases where both the operands have structure or union type, the result will have that type.

In cases where both operands have `void` type, the result will have `void` type.

In cases where both the second and third operands are pointers, or one is a null pointer constant and the other is a pointer, the type of the result is a pointer to a type qualified with all the type qualifiers of the types pointed-to by both the second and third operands.

In cases where both operands are pointers to compatible types or are differently qualified versions of a compatible type, the type of the result is the composite of the types.

In cases where one operand is a null pointer constant, the type of the result is the type of the other operand.

In cases where one operand is a pointer to `void` or is a qualified version of `void`, then the other operand is converted to type pointer to `void` and the result will have that type.

Conditional Syntax

conditional-expression:
logical-OR-expression
logical-OR-expression ? *expression* : *conditional-expression*

Assignment Operators

An assignment operator stores the value of the right expression into the object designated by the left operand, which must be a modifiable lvalue. The resulting assignment expression has the value of the left operand. The result is not an lvalue. Operands can be evaluated in any order.

The type of an assignment expression is the same as the type of its left operand. However, if the left operand has qualified type, then the assignment expression will have the type of the unqualified version of the type of the left operand.

There is a sequence point before and after the left operand is updated.

Simple Assignment

The simple assignment (=) operator will perform the following:

- convert the value of the right operand to the same type as the assignment expression (see the preceding section)
- store that value in the object designated by the left operand

If an object accesses the value stored in another object and the storage of the two objects overlaps at any location, then both of the following must be true, or the behavior is undefined:

- the overlap must be exact
- the two objects must have qualified or unqualified versions of a compatible type

The operands must meet one of the following conditions:

- The left operand must have qualified or unqualified arithmetic type and the right must have arithmetic type.
- The left operand must have a qualified or unqualified version of a structure or union type compatible with the type of the right.
- Both operands must be pointers to qualified or unqualified versions of compatible types; the type pointed to by the left must have all the qualifiers of the type pointed to by the right.
- One operand must be a pointer to an object or incomplete type and the other must be a pointer to a qualified or unqualified version of `void`; the type pointed to by the left must have all the qualifiers of the type pointed to by the right.
- The left operand must be a pointer and the right must be a null pointer constant.

Compound Assignment

In the following example of a compound assignment expression, the lvalue `E1` is evaluated only once:

```
E1 op = E2
```

whereas, in the corresponding simple assignment expression,

```
E1 = E1 op (E2)
```

the lvalue `E1` is evaluated twice.

The following compound assignment operators

```
+=
```

and

```
-=
```

require that the operands meet one of the following conditions:

- The left operand must be a pointer to an object type and the right must be an integral type.
- The left operand must have qualified or unqualified arithmetic type and the right must have an arithmetic type.

All other compound assignment operators require that the operand have the arithmetic type that is allowed by the corresponding binary operator.

Assignment Syntax

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of
`= *= /= %= += -= <<= >>= &= ^= =`

Comma Operator

When the comma operator is used, the evaluation takes place in the following order:

- the left operand is first evaluated as a `void` expression
- a sequence point is encountered
- the right operand is evaluated

The result is not an lvalue and has the same type and value as the right operand.

Comma Syntax

expression:
assignment-expression
expression , assignment-expression

Comma Operator Example

The comma can be used as a punctuator instead of an operator; for example, to separate arguments or initializers within a list. In those cases, the features of the comma that are described in the preceding section do not apply.

However, the comma can also be used within a parenthesized expression or within the second expression of a conditional operator.

For example, in the following function call, the function has three arguments and the second argument has a value of 5.

```
f(a,  (t=3,  t+2),  c)
```

The comma operator is very useful when used in conjunction with the conditional operator. For example:

```
#define igetc (f)  ((f) -> chars_left > 0 ?
                    ((f) -> chars_left --;
                     *(f) -> current_char++):
                    ifilup(f))
```

Chapter 10: Constant Expressions

Chapter 10: Constant Expressions

Overview

A constant expression can be used in the same way as a constant because it is an expression that can be evaluated at compile time rather than at execution time. The arithmetic precision and range is at least as great when the expression is evaluated in the translation phase as it would be if evaluated at run time.

Constant expressions evaluate to a constant (within the range of representable values for that constant type) using the same semantic rules as for non-constant expressions. For example, the following initialization contains a valid integral constant expression that evaluates to 16:

```
static int i = 3 * 5 + sizeof(char);
```

The following circumstances require a valid integral constant expression:

- the specification of the size of a bit-field member of a structure
- an enumeration constant value
- an array size
- a case constant value

Constant Expression Syntax

constant-expression:
conditional-expression

Integral Constant Expressions

An integral constant expression is of integral type and can have the following kinds of operands:

- integer constants

- enumeration constants
- character constants
- `sizeof` expressions
- floating constants that are cast operands

Cast operators must convert arithmetic types to integral types when they are used in integral constant expressions. The exception to this rule is when the cast operator is used as part of an operand to the `sizeof` operator.

Initializer Constant Expressions

Constant expressions in initializers must evaluate to one of the following:

- an arithmetic constant expression
- a null pointer constant
- an address constant
- an object type address constant plus or minus an integral constant expression

Arithmetic Constant Expressions

Arithmetic constant expressions are of arithmetic type and can have the following kinds of operands:

- integer constants
- floating constants
- enumeration constants
- character constants
- `sizeof` expressions

Address Constants

An address constant is a pointer. It points to either an lvalue designating a static object or a function designator. The address constant must be created explicitly by using the unary & operator or implicitly by the use of an array or function type expression.

The following operators may be used to create an address constant, but these operators cannot be used to access the value of an object.

- array-subscript `[ ]`
- member-access `.` and `->` operators
- the address `&` and indirection `*` unary operators
- pointer casts

Constant Expressions Constraints

The following operators are not allowed in constant expressions:

- assignment
- increment
- decrement
- function-call
- comma operators

However, these operators may be used when they are contained within the operand of a `sizeof` operator because such operands are not evaluated. (For example, the evaluation of `sizeof (i++)` does not result in `i` being incremented.)

Chapter 11: Statements

Chapter 11: Statements

Overview

Unlike an expression, a statement does not have a value, but rather specifies an action that will be executed. These actions, for the most part, are used to affect the flow of control of a program. Statements are generally executed in sequence, with exceptions as described in this chapter.

Statement Syntax

statement:
labeled-statement
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

Full Expressions

A full expression is defined as an expression that is not part of another expression. There is a sequence point at the end of a full expression.

A full expression can be any of the following:

- an initializer
- the expression in an expression statement
- the controlling expression of a selection statement (`if` or `switch`)
- the controlling expression of a `while` or `do` statement
- each of the three expressions (which are optional) in a `for` statement
- the expression (which is optional) in a `return` statement

Labeled Statements

A labeled statement can be any statement that is preceded by a label name, which is an identifier followed by a colon. The label names themselves do not affect processing.

The special `case` or `default` labels, as illustrated in the following syntax, may appear only in a `switch` statement (see the section "The `switch` Statement" later in this chapter).

Labeled Statement Syntax

labeled-statement:
    identifier : *statement*
    `case` *constant-expression* : *statement*
    `default` : *statement*

Compound Statement

A compound statement, usually referred to as a block, is used to group a set of statements into one syntactic unit.

That syntactic unit may have its own set of declarations and initializations. (See the section "Storage Durations of Objects" in Chapter 6.)

Objects that have automatic storage duration will have their initializers evaluated. The resulting values are then stored in those objects in the same order in which their declarators appear.

Compound Statement Syntax

compound-statement:
    { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
    declaration
    declaration-list declaration

statement-list:
    statement
    statement-list statement

Expression Statements

Expression statements are evaluated for their side effects, which may include assignments and function calls.

Expression Syntax

expression-statement:
expression$_{opt}$;

Null Statement Example

A null statement consists only of a semicolon and performs no operations.

For example, in the following program fragment, an empty loop body is supplied to the iteration statement by way of a null statement.

```
char *s;
/*...*/
while (*s++ != '\0')
  ;
```

Selection Statements

A selection statement conditionally selects among a set of statements. The value specified by a controlling expression will determine the particular statement(s) selected.

Selection Statement Syntax

selection-statement:
if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*
switch (*expression*) *statement*

The `if` Statement

The controlling expression of an `if` statement must have scalar type.

As illustrated in the preceding syntax, there are two forms of the selection statement that use the `if` statement.

In both forms, if the expression is not equal to 0 (that is, is true), then the first substatement is executed.

In the form that uses both `if` and `else`, the second substatement is executed if the expression compares equal to 0 (that is, is false). However, the second substatement is not executed if the first substatement is reached by way of a label.

The `else` in a selection statement is associated with the closest preceding `if`, as long as there is no other corresponding `else` for that `if` in the same block, which is not an enclosed block.

The `switch` Statement

A `switch` statement causes control to jump to, into, or past the statement called the `switch` body.

The controlling expression of a `switch` statement must have integral type. The controlling expression will have the integral promotions performed on it.

Each `case` label has its expression, which must be an integral constant expression, converted to the controlling expression's promoted type.

The particular action of the `switch` statement is specified by one or more of the following conditions:

- The value of the `switch` controlling expression, which must have integral type.
- The presence or absence of the `default` label. There can be no more than one `default` label in a `switch` statement. However, a `switch` statement that is enclosed within another `switch` statement may have a `default` label, or may have `case` constant expressions whose values duplicate any `case` constant expressions contained by the enclosing `switch` statement.

- The values of any `case` labels on or in the `switch` body.

  (A `case` or `default` label can be only located within the nearest enclosing `switch` statement.)

When the value of the converted `case` constant expression matches the value of the promoted controlling expression, control jumps to the statement following the matched `case` label. If the values do not match and there is a `default` label, control jumps to the labeled statement.

If the values do not match and there is no `default` label, then no part of the `switch` body is executed.

After conversion, each `case` constant expression in the same `switch` statement must have a unique value.

The implementation may place limits on the number of `case` values that can appear in a `switch` statement, as discussed in Chapter 1. (See Appendix C, item 12.i, in the *LPI-C User's Guide*.)

`switch` Statement Example

In the following program fragment, the object whose identifier is `d` is never initialized, although it exists with automatic storage duration within the block. Thus, if the value of the controlling expression is nonzero, the call to the function `f` will access an indeterminate value.

```
switch (integral_expr)
{
     double d = 0.0;
     case 0:
          d = 1.2;
          break;
     case -1;
          d = 3.4; /*fall through
                     to default*/
     default:
          f(d);
}
```

Iteration Statements

An iteration statement causes a statement (the loop body) to be executed repeatedly as long as the resulting value of the controlling expression is not equal to 0 (that is, true). The controlling expression must have scalar type.

Iteration Syntax

iteration-statement:

while (*expr*) *statement*

do *statement* **while** (*expr*) ;

for ($expr_{opt}$; $expr_{opt}$; $expr_{opt}$) *statement*

The `while` Statement

When the **while** statement is used for iteration, the evaluation of the controlling expression takes place before each execution of the loop body.

The `do` Statement

When the **do** statement is used for iteration, the evaluation of the controlling expression takes place after each execution of the loop body.

That is, the loop is always executed at least once.

The `for` Statement

Statement 1 is exactly equivalent to the subsequent sequence of statements (Statements 2), as long as a **continue** is not present within the loop body:

Statement 1

```
for ( expression-1 ; expression-2 ; expression-3 ) statement
```

Statement 2

```
expression-1 ;
while (expression-2) {
      statement
   expression-3;
}
```

Thus, in a `for` statement:

expression-1 specifies initialization for the loop

expression-2 is the controlling expression and specifies an evaluation made before each iteration, with the result that execution of the loop terminates when the expression compares equal to 0 (that is, false)

expression-3 specifies an operation (for example, decrementing) that is performed after each iteration

Neither *expression-1* nor *expression-3* is required to be present.

If *expression-2* is omitted, it is replaced by a nonzero constant.

Jump Statements

A jump statement transfers control unconditionally. (That is, it causes an unconditional jump to elsewhere in the translation unit.)

Jump Statement Syntax

jump-statement:

```
goto identifier ;
continue ;
break ;
return expression_opt ;
```

Jump Statement Example

The user may find it necessary to jump into a set of statements. The following example presents one possible approach to a situation such as this, based on the following assumptions:

- The general initialization code accesses objects whose scope is limited to the current function.
- The general initialization code is too large to be duplicated.
- The code that determines the subsequent operation (for example, a continue statement) must be located at the head of the loop.

```
/*...*/
goto first_time;
for (;;) {
     /* determine next operation */
     if (need to reinitialize) {
          /* reinitialize-only code */
     first_time:
          /* general initialization code */
          continue;
     }
     /* handle other operations */
     }
```

The `goto` Statement

A `goto` statement transfers control to the labeled statement in the enclosing function. That label is named by the identifier in the `goto` statement.

The `continue` Statement

A `continue` statement, which may only appear as part of the loop body for the `while`, `do`, or `for` statements, transfers control to the end of that loop body to terminate the current iteration and initiate the next one.

For example, in each of the following statements, the `continue` is equivalent to `goto end_loop;`. (A null statement follows the `end_loop:` label.)

If the `continue` statement shown is in an enclosed iteration statement, it is interpreted within that statement.

```
while (expr){   do {                for (;expr;) {
   /*code*/        /*code*/            /*code*/
   continue;      continue;           continue;
   /*code*/      /*code*/             /*code*/
   end_loop: ;    end_loop: ;         end_loop: ;
}               } while (expr);     }
```

The `break` Statement

A `break` statement may appear as a part of a `switch` body or `while`, `do`, or `for` loop body and is used to terminate execution of the smallest enclosing `switch`, `while`, `do`, or `for` statement.

The `return` Statement

A `return` statement is used to terminate the execution of a function and return control (and, optionally, a value) to the function's caller.

A `return` statement can contain expressions, and there can be more than one `return` statement in a function. (However, a `return` statement with an expression may not appear in a function whose return type is `void`.)

After the execution of a `return` statement that contains an expression, the value of that expression is returned to the caller as the value of the function call expression. (If the expression type is different from the type of the function, the expression type is converted as if by assignment to the type of the function.)

The behavior is undefined if, after the execution of a `return` statement that does not contain an expression, the caller uses the return value of the function call.

In a case in which the compilation reaches the right bracket (}) that terminates a function, the result is the same as though the `return` statement without an expression were executed.

Chapter 12: ANSI C Compatibility Issues

Chapter 12: ANSI C Compatibility Issues

Overview

This chapter describes changes introduced to the C language by the new ANSI standard and LPI extensions to ANSI C.

ANSI C Changes

In developing the new ANSI standard for C, the ANSI committee took into consideration the fact that considerable amounts of C code have been written since the C language was first introduced as part of the development of the UNIX operating system. Some of these changes could cause valid "old-style" programs (Kernighan and Ritchie or other implementations) to perform differently. The committee attempted, wherever possible, to avoid "quiet changes" that would cause a valid program to perform differently without notice. In the interest of maintaining maximum compatibility and portability, the programmer should be aware of the degree to which software written in pre-ANSI C code can be moved from one computer system to another.

The following is an overall summary of some of the major differences between the old K&R definition of C and the new ANSI C.

- The "integral promotion" rules have changed from the old-style "unsigned-preserving" method to the ANSI "value-preserving" method. The ANSI committee considered this to be the most serious change made to C. (See the following "Integral Promotions" section.)
- A standard run-time library and associated header files have been defined.
- Preprocessing is much more carefully defined in ANSI C. This has been an area in which many older K&R-based C implementations differed significantly. Some of the more notable ANSI C preprocessing features are:
  - A hierarchy of translation phases is specified and preprocessing is explicitly token-based. This clarifies a number of previous ambiguities in preprocessing and lexical analysis.

- White-space is forbidden within tokens; for example, "`+ =`" (note the white-space) is now illegal.
- A new and powerful "stringize" preprocessing operator `#` has been added for the creation of string literals from a token sequence in a macro function argument.
- A new and powerful "token-paste" preprocessing operator `##` has been added to concatenate two tokens into a single token.
- A new and much-needed preprocessing control directive `#elif` has been added to help simplify complicated `#if/#else/#if` conditional compilation constructs.
- A new preprocessing unary operator `defined(`*id*`)` has been added which evaluates to `1` if the specified identifier *id* is currently defined; otherwise it evaluates to `0`. This is primarily to be used within the controlling expression of an `#if/#elif` conditional compilation directive.
- A new `#pragma` preprocessing directive has been added to provide a standard way of communicating implementation-defined directives to the compiler.
- A new `#error` preprocessing directive has been added to provide a way to explicitly generate a preprocessing error message.
- Macros are now expanded within `#include` and `#line` directives.
- Macros may now be undefined using the `#undef`ined directive.
- Macros may be redefined only if the definitions are identical.
- Macro parameter substitution is not performed within character constants and string literals.
- The rules for macro expansion have been greatly clarified and tightened up; recursive macro definitions are now well-defined.
- Trailing text on `#else` and `#endif` directives is not permitted.
- The pre-defined macros `__STDC__`, `__LINE__`, `__FILE__`, `__DATE__`, and `__TIME__` have been defined.
- Comments are replaced by one space character during lexical analysis.

- Trigraph sequences (introduced by a `??`) have been added to allow for the representation of characters that are lacking in some character sets. Trigraphs for `#`, `\`, `^`, `[`, `]`, `{`, `}`, `|`, and `~` have been defined.
- The new escape sequences `\a`, `\?`, and `\x`*hex-digits* have been added for use within character constants and string literals.
- The integer constant suffixes `U` (or `u`) and `F` (or `f`) have been added for `unsigned` and `float` type coercion and may be used in conjunction with the `L` (or `l`) suffix.
- For internationalization purposes, syntax and semantics have been added to deal with "wide" character constants and string literals (for example, string literals containing Kanji characters).
- The antiquated "=+" style compound assignment operators (that is, rather than "+=") are now truly illegal.
- The minimum number of significant characters that must be recognized for internal identifiers has been extended from 6 to 31.
- Structures and unions may be assigned to each other, passed as arguments to functions, and returned by functions.
- Unions may now be initialized (the initializer refers to the first member).
- Automatic structures, unions, and arrays may now be initialized.
- Character arrays with an explicit size may be initialized with a string literal with exactly that many characters, thereby allowing the null byte (`\0`) not to be included.
- Modifying the contents of string literals is now illegal and yields undefined behavior.
- A `const` type qualifier has been added to specify that a data object (and/or a pointer to a data object) is constant and should not be modified by the program (it may of course be initialized in its declaration). All obvious attempts to do so will be detected and diagnosed by the compiler, and such objects could even be placed in a read-only area of memory.
- A `volatile` type qualifier has been added. The use of this qualifier is not as obvious as the use of the qualifier `const`. `volatile` specifies that a data object may be modified by some entity not known to the compiler and outside direct control of the

program; for example, a hardware device or an alternate thread of a multi-tasking program. References to objects declared as `volatile` will never be "optimized-out."

- A `signed` type specifier has been added, primarily to be used to guarantee a `char` data type is signed (that is, `signed char`).
- Restrictions on type (especially pointer type) compatibility have been tightened up and added. As with function prototypes, these changes should greatly increase the maintainability and portability of programs.
- Structure and union member names are in their own name space and need to be unique only within the structure or union, rather than across the entire compilation unit.
- Labels are now in their own name space.
- Proper scoping rules for `extern` declarations within block scope are now enforced.
- A declaration of only a structure or union tag redeclares that tag even if it was declared in an outer scope.
- Adjacent string literals are now concatenated at compile time, thus providing a way to neatly use very long string literals.
- The enumerated data type `enum` has been added (similar to Pascal enumerated types).
- A `void` data type has been added, which can be used to define a function which returns no value, to define a (prototyped) function which accepts no arguments, or as a generic pointer type.
- Perhaps the most important and certainly the most visible change in ANSI C is the addition of function "prototype" declarations, and a new syntax for function definition declarators. This allows a function to be declared along with the type of each of its arguments; the number of arguments is also implicitly specified. This will aid the compiler immensely in detecting and providing helpful diagnostics when the number or type, or both, of arguments in functions calls are not correct (that is, when they do not match the function prototype declaration). Also, the notion of a "prototype scope" has been introduced. Old-style (K&R) function declarations and function definitions are still supported by ANSI C, with some restrictions concerning their use.

In addition, a standard method of declaring and defining functions accepting a variable number of arguments has been added. These additions should significantly decrease development time, and increase program reliability, maintainability, and portability.

- Integral constant typing rules have been clarified and changed slightly.
- Within expressions, `float` types are no longer required to be promoted to type `double`.
- A unary `+` operator has been added (for symmetry with the unary `-` operator).
- A pointer to a function may be used as a function designator without an explicit de-reference (`*`) operation.
- Creating a pointer to just beyond the end of an array has been legalized.
- The address of an array may now be properly taken using the `&` operator.
- The address of an object with `register` storage class may not be taken.
- The controlling expression (and the `case` labels) of a `switch` statement may have any integral type (not just `int`).
- The `sizeof` operator now returns a value (representing the size in bytes of its operand) of type `size_t` rather than of type `int`; the type `size_t` is implementation-defined in a standard header file (`<stddef.h>`).
- Minimum/maximum compilation and numerical limits have been defined.

Integral Promotions

Probably the most serious semantic change introduced into ANSI C involves the way in which shorter unsigned integral types (that is, `unsigned char`, `unsigned short`, or small `unsigned int` bit-fields) are promoted when used in expressions. These "integral promotions" are applied to the operands of nearly all operations involving shorter unsigned integral types (either directly or in the process of applying the "default argument promotions" or the "usual arithmetic conversions").

The problem arises from the fact that many older C implementations used different rules in applying the integral promotions; they used what is referred to as "unsigned-preserving" rules, whereas ANSI C uses what is referred to as "value-preserving" rules. The unsigned-preserving rules specify that a shorter unsigned integral type should be promoted to `unsigned int`. The value-preserving rules specify that a shorter unsigned integral type should be promoted to `int` if it will fit (that is, if its size in bits is less than or equal to that of the size in bits of an `int`) and to `unsigned int` if not.

The two tables below summarize the integral promotions for old-style C and ANSI C:

TABLE 12-1 Old C Unsigned-Preserving Integral Promotions

OPERAND TYPE	OPERAND SIZE	PROMOTED TYPE
`char`	any	`int`
`short`	any	`int`
`int`	any	`int`
`unsigned char`	any	`unsigned int`
`unsigned short`	any	`unsigned int`
`unsigned int`	any	`unsigned int`

TABLE 12-2 ANSI C Value-Preserving Integral Promotions

OPERAND TYPE	OPERAND SIZE	PROMOTED TYPE
`char`	any	`int`
`short`	any	`int`
`int`	any	`int`
`unsigned char`	smaller than `int`	`int`
`unsigned char`	same as `int`	`unsigned int`
`unsigned short`	smaller than `int`	`int`
`unsigned short`	same as `int`	`unsigned int`
`unsigned int`	smaller than `int`	`int`
`unsigned int`	same as `int`	`unsigned int`

Note
The size of an `unsigned int` could be smaller than that of an `int` when it is a bit-field. For example:

```
struct { unsigned int x: 3; };
```

(Note that you may not apply the `sizeof` operator to a bit-field.)

The difference in these rules may be somewhat subtle, and fortunately, in the vast majority of cases it will cause no problem. But the differences can create ambiguities in the semantics of certain kinds of expressions, thereby creating the possibility for a "quiet change," which can lead to insidious and hard-to-find bugs.

Note
The remainder of this discussion assumes a "typical" two's complement implementation.

When an expression possesses all of the following attributes, the result will be an ambiguous, "questionably signed" value:

- The expression involves a shorter unsigned integral type operand (that is, `unsigned char` or `unsigned short`).
- The expression produces an `int`-wide result (that is, the integral promotions are performed on the operand because of a unary or binary operation in which the other operand is an `int` or shorter type).
- The sign bit of the result is set.

The result is a questionably signed value, because if the unsigned-preserving integral promotion rules were applied, then the result would be of type `unsigned int` and would contain a very large positive value. But if the value-preserving integral promotion rules were applied, then the result would be of type `int` and would contain a negative value.

A questionably signed value does not always cause problems; it will cause problems when at least one of the following is true:

- It is the left operand of a right-shift operator (`>>`), in an implementation where a right-shift is arithmetic rather than logical.
- It is an operand of a divide (`/`), a remainder (`%`), or a relational (`<`, `<=`, `>`, `>=`) operator.

For example, the following could be used as a test to determine which method of integral promotions an implementation uses:

```
int             i = -9;
unsigned short us =  3;

if ((i / us) < 0)
     printf ("Value-preserving (ANSI C)\n");
else printf ("Unsigned-preserving (old C)\n");
```

Here, `us` is a shorter unsigned integral type, which is promoted to an `int`-wide result. It is promoted because it is an operand of a division operator (`/`) which specifies that the usual arithmetic conversions (which include the integral promotions) be performed on its operands. Specifically, it is promoted either to `int` (if value-preserving rules are applied) or to `unsigned int` (if unsigned-preserving rules are applied). If the value-preserving rules are applied and `us` is promoted to `int`, then since `i` is also an `int`, a signed integer division will be performed, and a signed negative integer will result (that is, −9/3 giving −3). But, if the unsigned-preserving rules are applied and `us` is promoted to `unsigned int`, then by the rules of the arithmetic conversions, `i` is also promoted to `unsigned int`, an unsigned integer division will be performed, and a large unsigned value will result (that is, 4294967287/3 giving 1431655762, assuming an `int` is 32 bits).

In conclusion, mixing signed and unsigned integral types can be confusing and dangerous, and it should be done only with great care and appropriate use of casts. Such an important change would warrant a compile-time switch on any ANSI C implementation, which would cause the integral promotions to be performed using the old-style C unsigned-preserving rules rather than the ANSI C value-preserving rules. LPI-C does provide such a capability as well as other compatibility switches. Refer to your *LPI-C User's Guide* for more information.

Integral Constant Typing

Another problem, very much related to the unsigned-preserving versus value-preserving conflict discussed above, may arise when assigning types to integral constants.

The problem arises from the fact that many older C implementations essentially will not assign the type `unsigned long` to integral

constants, whereas ANSI C will, if necessary. Thus, in certain circumstances an integral constant could be questionably signed, and when used in certain kinds of expressions could be semantically ambiguous, thereby creating the possibility for a "quiet change." The two tables below summarize the integral constant typing rules for old-style C and ANSI C:

TABLE 12-3 Old C Integral Constant Typing Rules

INTEGRAL CONSTANT	ASSIGNED TYPE (first in list which fits)
Unsuffixed decimal:	`int, long`
Unsuffixed octal/hex:	`int, unsigned int, long`
L suffixed:	`long`

TABLE 12-4 ANSI C Integral Constant Typing Rules

INTEGRAL CONSTANT	ASSIGNED TYPE (first in list which fits)
Unsuffixed decimal:	`int, long, unsigned long`
Unsuffixed octal/hex:	`int, unsigned int, long, unsigned long`
L suffixed:	`long, unsigned long`
U suffixed:	`unsigned, unsigned long`
L & U suffixed:	`unsigned long`

For example, the following could be used as a test to determine which method of integral constant typing rules an implementation uses (assuming a two's complement 32-bit implementation).

```
if    (2147483648 > 0)
         printf ("ANSI C integral constant
                  typing rules\n");
else     printf ("Old C integral constant
                  typing rules\n");
```

In addition to the possibility of this kind of compatibility problem, portability problems may crop up when using integral constants, even

when using solely an older C or ANSI C implementation. For example, consider the function call "`f(65000);`" with the function `f` defined as taking an `int` argument. In implementations in which an `int` is 32 bits, `65000` will be of type `int` and all is well. In implementations in which an `int` is 16 bits, however, `65000` will be of type `long`, 4 bytes will be passed (assuming a `long` is 32 bits), `f` will be expecting 2 bytes, and an insidious bug will probably result.

In conclusion, great care should be taken when using integral constants when the size or type, or both, of the constant is important, and should explicitly be typed either by the `U` and/or `L` suffix or by appropriate use of casts.

Again, such a change would warrant a compile-time switch on an ANSI C implementation, which would cause the integral constant typing to be performed according to old C rules rather than the ANSI C rules. LPI-C does provide such a capability, as well as other compatibility switches. Refer to your *LPI-C User's Guide* for more information.

LPI-C Extensions

In order for LPI-C to be able to compile and execute correctly, with minimal change, the vast amounts of C code that exist, and to be compatible with older pre-ANSI C implementations, a number of compatibility modes are supported. Listed below are some of the key non-ANSI or undefined-ANSI extensions that are supported by LPI's conforming ANSI C implementation. These extensions are provided by way of compiler options for non-ANSI items, or as default behavior for undefined-ANSI items. Refer to your *LPI-C User's Guide* for details on how to turn on these options.

- Optionally undefines `__STDC__` (intending to indicate a non-conforming ANSI C implementation).
- Optionally ignores trigraph sequences (that is, no trigraph sequence mapping will be performed).
- Optionally accepts `long float` as a synonym for `double` in a declarator.
- Optionally promotes all `float` types to type `double` in expressions.

- Optionally recognizes only old-style C escape sequences within string literals and character constants (that is, the `\a`, `\?`, and `\x`*hexadecimal-digits* escape sequences will not be recognized). When any unrecognized escape sequences are encountered, a warning will be given, and compilation will continue as if the backslash in the escape sequence were not present.
- Optionally uses old-style C file scoping rules for functions and data objects declared as `extern` within an inner block scope. That is, functions and data objects declared as `extern` within an inner block will actually be declared as if the declaration had appeared outside of any block (that is, at file scope).
- Optionally assigns types to integer constants by old-style C rules rather than ANSI C rules. (See Tables 12-3 and 12-4.)
- Optionally applies the integral promotions of integral types in expressions according to the old-style C "unsigned-preserving" rather than ANSI C "value-preserving" rules. (See Tables 12-1 and 12-2.)
- Optionally accepts the type `char *` as equivalent to the generic pointer type `void *`, without a warning.
- Optionally allows additionally the integral types `char`, `short`, and `long` (signed or unsigned) to be declared as bit-fields, with appropriate maximum widths, without a warning.
- Optionally allows the destructive redefinition of macros, with a warning. The default ANSI C behavior in this situation is to emit a warning and ignore the redefinition.
- Optionally allows innocuous redefinitions of function-like macros, which differ only in the spelling of formal macro parameter identifiers, without a warning. For example:

```
#define min(q,r)  (((q) <=  (r)) ?  (q) : (r))
#define min(x,y)  (((x) <=  (y)) ?  (x) : (y))
```

  would be silently accepted, without warning.
- Optionally allows trailing text on a `#else` or `#endif` preprocessing directive, without a warning.

- Optionally allows the expansion of function-like macro invocations with empty arguments as if the (empty) arguments consisted of no tokens, without a warning. For example,

```
#define minus(x,y)   (x-y)
minus(,123)
```

  would yield "(-123)." ANSI C leaves the behavior of such a construct undefined.

- Optionally emits warnings wherever a function-like macro is defined such that old-style C formal macro parameter substitution within string literals could be performed.
- Optionally causes old-style C formal macro parameter substitution within string literals to be performed.
- Optionally emits warnings wherever a function-like macro is defined such that old-style C formal macro parameter substitution within character constants could be performed.
- Optionally causes old-style C formal macro parameter substitution within character constants to be performed.
- Optionally emits a warning for each unrecognized `#pragma` preprocessing directive; by default they are completely ignored.
- Optionally ignores non-ANSI `#ident` preprocessing directives completely without warning.

Compatibility Options

For a list of compatibility options with older pre-ANSI implementations (most notably PCC-based compilers), refer to your *LPI-C User's Guide*.

Glossary

ANSI	American National Standards Institute, which defines standards for programming languages.
ASCII	American National Standard Code for Information Interchange, which is the standard for defining the representation of character data.
abstract semantics	Conceptual steps to be taken in the program's execution.
actual semantics	Steps as they are taken in executing a compiled program. Actual semantics may differ from abstract semantics and produce the same result.
address	The storage location of a value in memory.
alignment requirements	Implementation-defined restrictions on addresses of specified data types so that the addresses are divisible by specified integers.
argument	An expression defining a value to be passed to a called function.
array	An ordered set of values, each having the same data types.

behavior — The manner in which an implementation reacts to a certain construction. (See **implementation-defined behavior, locale-specific behavior, undefined behavior,** and **unspecified behavior.**)

bit — The storage unit in the execution environment that is able to hold an object with one of two values.

bit-field — A data object that consists of a specified number of bits. A bit-field must be no larger than an `int`; it is treated as an `unsigned int` or `signed int`.

byte — A group of adjacent bits, the number of which is implementation-defined (although typically eight), constituting a storage unit that is able to hold any member of the basic character set of the execution environment.

character — The basic indivisible data unit of the language. See **byte.**

comment — Any part of a source line in a program, command, or file that serves as documentation instead of as an instruction.

compilation — Translation of a source program into an executable program.

compilation unit — See **translation unit.**

compiler — A program that translates a source program into an executable program.

constraints — Restrictions that language syntax and semantics place upon the interpretation of language elements.

default argument promotions	Data type conversions that take place when arguments are passed to a function without a function prototype.
diagnostic message	A message identifying some error condition that has been encountered.
external linkage	Connection that instances of an identifier have across multiple translation units.
function	A routine defined to calculate or to transform variable sets of data within given parameters. A function is called by its name and (optionally) a list of values (arguments) upon which it is to operate.
hexadecimal	Pertaining to a numbering system with a base of 16, in which digits 0 through 9 and the characters A through F and a through f (representing 10 through 15) are valid digits.
identifier	A combination of letters, digits, and underscores (_) that constitutes a data name.
implementation	A set of software that, in a specific translation environment, translates programs for a specific execution environment and supports the execution of functions in that environment.
implementation-defined behavior	Response that a translation program itself determines (and needs to document). An example of implementation-defined behavior is the propagation of high-order bit when a signed integer is shifted right.

implementation limits	Restrictions that a specific implementation imposes upon programs.
integral promotion	Conversion of a smaller data type to `int` or `unsigned int` when it is used as an operand in an expression.
internal linkage	Connection that instances of an identifier have within one translation unit.
keyword	A token that is reserved for use by the language for syntactic and semantic purposes; keywords must not be used as variable names.
label	An identifier that designates a specific statement in an LPI-C program.
least significant bit	The low-order bit in a data object.
linkage	The connection between multiple instances of an identifier across blocks of code and among source files, object modules, and libraries.
locale-specific behavior	Computer response that an implementation makes dependent upon local conventions of nationality, culture, and language. An example of locale-specific behavior is the return value for the `islower` function when the arguments are characters other than the 26 lower-case English letters.
most significant bit	The high-order bit in a data object.

multibyte character	A member of the extended character set of either the source or the execution environment, including characters that need more than one byte for their representation.
object	A storage unit that contains one or more values in the execution environment and, except in the case of bit-fields, consists of a byte or group of adjacent bytes whose number, order, and encoding are either explicitly specified or implementation-defined. Referenced objects are interpreted according to their data types.
octal	Pertaining to a numbering system with a base of 8, in which the digits 0 through 7 are the only valid numbers.
old-style function	Function definition that lists the parameter names and parameter types separately.
parameter	An object that is declared in a function declaration or definition and acquires a value on entry to the function.
pointer	A variable whose value is an address.
portability	The ability of software to be compiled and run in more than one environment.
program	A set of instructions that directs the computer to perform a series of tasks in a specified order.
prototyped function	Function definition that specifies the number and types of parameters it accepts, as well as its return types.

qualified type

Identifier that has special properties added to its type, indicated by one of the following qualifiers: `const` (object is not modifiable by the program), or `volatile` (object may be changed by an entity that the implementation does not know or does not control.) Qualifiers must be the same for data types to be compatible.

reserved word

See **keyword**.

scope

The region of the program text in which a name can be referenced.

sequence point

A point within program execution at which the results of the actual semantics must match those of the abstract semantics.

source code

The original form of a program, before translation by a compiler.

translation

The process of converting the source code of the program into executable code.

translation unit

The basic piece of a source program that is compiled into executable form; it consists of the source file, its headers (source text inserted by the `#include` directive), excluding source lines omitted by any of the conditional inclusion preprocessing directives.

undefined behavior

Behavior that applies to the use of incorrect program constructs or data, for which ANSI C imposes no requirements. An example of undefined behavior is the behavior on integer overflow.

unqualified type

A type that is not qualified by the keywords `const` or `volatile`. (See **qualified type**.)

unspecified behavior

Behavior, for a correct program construct and correct data, upon which ANSI C imposes no requirements. An example of unspecified behavior is the order in which the arguments to a function are evaluated.

usual arithmetic conversions

Process by which the operands of an operator in an arithmetic expression are converted to the same data type.

Appendix A: Language Syntax Summary

This appendix summarizes LPI-C language syntax.

Tokens

token:
- *keyword*
- *identifier*
- *constant*
- *string-literal*
- *operator*
- *punctuator*

preprocessing-token:
- *header-name*
- *identifier*
- *pp-number*
- *character-constant*
- *string-literal*
- *operator*
- *punctuator*
- each non-white-space character that cannot be one of the above

Keywords

keyword: one of

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Identifiers

identifier:
    nondigit
    identifier nondigit
    identifier digit

nondigit: one of

_	a	b	c	d	e	f	g	h	i	j	k	l	m
	n	o	p	q	r	s	t	u	v	w	x	y	z
	A	B	C	D	E	F	G	H	I	J	K	L	M
	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

digit: one of
    0 1 2 3 4 5 6 7 8 9

Constants

constant:
    floating-constant
    integer-constant
    enumeration-constant
    character-constant

floating-constant:
    fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
    digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
    digit-sequence$_{opt}$. *digit-sequence*
    digit-sequence .

exponent-part:
    e *sign*$_{opt}$ *digit-sequence*
    E *sign*$_{opt}$ *digit-sequence*

sign: one of
    + -

digit-sequence:
    digit
    digit-sequence digit

floating-suffix: one of

`f` `l` `F` `L`

integer-constant:

decimal-constant integer-suffix$_{opt}$

octal-constant integer-suffix$_{opt}$

hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:

nonzero-digit

decimal-constant digit

octal-constant:

`0`

octal-constant octal-digit

hexadecimal-constant:

`0x` *hexadecimal-digit*

`0X` *hexadecimal-digit*

hexadecimal-constant hexadecimal-digit

nonzero-digit: one of

`1` `2` `3` `4` `5` `6` `7` `8` `9`

octal-digit: one of

`0` `1` `2` `3` `4` `5` `6` `7`

hexadecimal-digit: one of

`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`

`a` `b` `c` `d` `e` `f`

`A` `B` `C` `D` `E` `F`

integer-suffix:

unsigned-suffix long-suffix$_{opt}$

long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of

`u` `U`

long-suffix: one of

`l` `L`

enumeration-constant:
identifier

character-constant:
´ *c-char-sequence* ´
L´ *c-char-sequence* ´

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any member of the source character set except
the single-quote ´, backslash \, or new-line character
escape-sequence

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of
`\´ \" \? \\`
`\a \b \f \n \r \t \v`

octal-escape-sequence:
\ *octal-digit*
\ *octal-digit octal-digit*
\ *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

String literals

string-literal:
" *s-char-sequence*$_{opt}$ "
L" *s-char-sequence*$_{opt}$ "

s-char-sequence:
s-char
s-char-sequence s-char

s-char:

any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence

Operators

operator: one of

```
[   ]   (   )   .   -> ++   --   &   *   +   -       !   sizeof
/   %   <<   >>   <   >   <=   >=   ==   !=   ^   |   &&   | |
?   :   =   *=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=
,   #   ##
```

Punctuators

punctuator: one of

```
[   ]   (   )   {   }   *   ,   :   =   ;   ...   #
```

Header names

header-name:

<h-char-sequence>
"q-char-sequence"

h-char-sequence:

h-char
h-char-sequence h-char

h-char:

any member of the source character set except
the new-line character and >

q-char-sequence:

q-char
q-char-sequence q-char

q-char:

any member of the source character set except
the new-line character and "

Preprocessing numbers

pp-number:
    digit
    . *digit*
    pp-number digit
    pp-number nondigit
    pp-number e *sign*
    pp-number E *sign*
    pp-number .

Expressions

primary-expression
    identifier
    constant
    string-literal
    (*expression*)

postfix-expression
    primary-expression
    postfix-expression [*expression*]
    postfix-expression ($argument\text{-}expression\text{-}list_{opt}$)
    postfix-expression . *identifier*
    postfix-expression -> *identifier*
    postfix-expression ++
    postfix-expression --

argument-expression-list:
    assignment-expression
    argument-expression-list , *assignment-expression*

unary-expression:
    postfix-expression
    ++ *unary-expression*
    -- *unary-expression*
    unary-operator cast-expression
    sizeof *unary-expression*
    sizeof (*type-name*)

unary-operator: one of
    & * + - !

cast-expression:

unary-expression

(*type-name*) *cast-expression*

multiplicative-expression:

cast-expression

multiplicative-expression * *cast-expression*

multiplicative-expression / *cast-expression*

multiplicative-expression % *cast-expression*

additive-expression:

multiplicative-expression

additive-expression + *multiplicative-expression*

additive-expression - *multiplicative-expression*

shift-expression:

additive-expression

shift-expression << *additive-expression*

shift-expression >> *additive-expression*

relational-expression:

shift-expression

relational-expression < *shift-expression*

relational-expression > *shift-expression*

relational-expression <= *shift-expression*

relational-expression >= *shift-expression*

equality-expression:

relational-expression

equality-expression = = *relational-expression*

equality-expression ! = *relational-expression*

bitwise-AND-expression:

equality-expression

bitwise-AND-expression & *equality-expression*

bitwise-XOR-expression:

bitwise-AND-expression

bitwise-XOR-expression ^ *bitwise-AND-expression*

bitwise-OR-expression:

exclusive-OR-expression

bitwise-OR-expression | *bitwise-XOR-expression*

logical-AND-expression:
    bitwise-OR-expression
    logical-AND-expression **&&** *bitwise-OR-expression*

logical-OR-expression:
    logical-AND-expression
    logical-OR-expression *logical-AND-expression*

conditional-expression:
    logical-OR-expression
    logical-OR-expression **?** *expression* **:** *conditional-expression*

assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression

assignment-operator: one of
    = *= /= %= += -= <<= >>= &= ^= =

expression:
    assignment-expression
    expression **,** *assignment-expression*

constant-expression:
    conditional-expression

Declarations

declaration:
    declaration-specifiers $init\text{-}declarator\text{-}list_{opt}$

declaration-specifiers:
    storage-class-specifier $declaration\text{-}specifiers_{opt}$
    type-specifier $declaration\text{-}specifiers_{opt}$
    type-qualifier $declaration\text{-}specifiers_{opt}$

init-declarator-list:
    init-declarator
    init-declarator-list **,** *init-declarator*

init-declarator:
    declarator
    declarator **=** *initializer*

storage-class-specifier:

typedef
extern
static
auto
register

type-specifier:

void
char
short
int
long
float
double
signed
unsigned
struct-or-union-specifier
enum-specifier
typedef-name

struct-or-union-specifier:

struct-or-union identifier$_{opt}$ { *struct-declaration-list* }
struct-or-union identifier

struct-or-union:

struct
union

struct-declaration-list:

struct-declaration
struct-declaration-list struct-declaration

struct-declaration:

specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:

type-specifier specifier-qualifier-list
type-qualifier specifier-qualifier-list

struct-declarator-list:

struct-declarator
struct-declarator-list, struct-declarator

struct-declarator:
    declarator
    declarator$_{opt}$: *constant-expression*

enum-specifier:
    enum *identifier*$_{opt}$ { *enumerator-list* }
    enum *identifier*

enumerator-list:
    enumerator
    enumerator-list , *enumerator*

enumerator:
    enumeration-constant
    enumeration-constant = *constant-expression*

type-qualifier:
    const
    volatile

declarator: *pointer*$_{opt}$ *direct-declarator*

direct-declarator:
    identifier
    (*declarator*)
    direct-declarator [*constant-expression*$_{opt}$]
    direct-declarator (*parameter-type-list*)
    direct-declarator (*identifier-list*$_{opt}$)

pointer:
    * *type-qualifier-list*$_{opt}$
    * *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier

parameter-type-list:
    parameter-list
    parameter-list , . . .

parameter-list:
parameter-declaration
parameter-list , *parameter-declaration*

parameter-declaration:
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
identifier
identifier-list , *identifier*

type-name:
specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
pointer
pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
(*abstract-declarator*)
direct-abstract-declarator$_{opt}$ [*constant-expression*$_{opt}$]
direct-abstract-declarator$_{opt}$ (*parameter-type-list*$_{opt}$)

typedef-name:
identifier

initializer:
assignment-expression
{ *initializer-list* }
{ *initializer-list* , }

initializer-list:
initializer
initializer-list , *initializer*

Statements

statement:
    labeled-statement
    compound-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement

labeled-statement:
    identifier : *statement*
    case *constant-expression* : *statement*
    default : *statement*

compound-statement:
    { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
    declaration
    declaration-list declaration

statement-list:
    statement
    statement-list statement

expression-statement:
    expression$_{opt}$;

selection-statement:
    if (*expression*) *statement*
    if (*expression*) *statement* **else** *statement*
    switch (*expression*) *statement*

iteration-statement:
    while (*expression*) *statement*
    do *statement* **while** (*expression*) ;
    for (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*

jump-statement:
    goto *identifier* ;
    continue ;
    break ;
    return *expression*$_{opt}$;

External Definitions

translation-unit:
    external-declaration
    translation-unit external-declaration

external-declaration:
    function-definition
    declaration

function-definition:
    declaration-specifiers$_{opt}$ *declarator*
    declaration-list$_{opt}$

Preprocessing Directives

preprocessing-file:
    group$_{opt}$

group:
    group-part
    group group-part

group-part:
    pp-tokens$_{opt}$ *new-line*
    if-section
    control-line

if-section:
    if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
    # if *constant-expression new-line group*$_{opt}$
    # ifdef *identifier new-line group*$_{opt}$
    # ifndef *identifier new-line group*$_{opt}$

elif-groups:
    elif-group
    elif-groups elif-group

elif-group:
    # elif *constant-expression new-line group*$_{opt}$

else-group:

 # else *new-linegroup*$_{opt}$

endif-line:

 # endif *new-line*

control-line:

#include *pp-tokens new-line*
#define *identifier replacement-list new-line*
#define *identifier lparen identifier-list*$_{opt}$ **)**
replacement-list new-line
#undef *identifier new-line*
#line *pp-tokens new-line*
#error *pp-tokens*$_{opt}$ *new-line*
#pragma *pp-tokens*$_{opt}$ *new-line*
new-line

lparen:

the left-parenthesis character without preceding white space

replacement-list:

pp-tokens$_{opt}$

pp-tokens:

preprocessing-token
pp-tokens preprocessing-token

new-line:

the new-line character

Appendix B: Identifier List

This appendix provides an alphabetical listing of all reserved ANSI C identifiers, as well as a list of identifiers that may be reserved for hidden internal use, may be reserved by the implementation, or may be reserved by the ANSI C Standard in the future.

ANSI C reserves many identifiers for its own use. Reserved identifiers, when in scope, should not be defined by the programmer (in the same name space); otherwise the identifiers may collide and the behavior will be undefined.

The following section contains an alphabetical listing of all of the identifiers reserved by ANSI C and where each is defined (if defined in a standard header file), as well as what kind of entity each identifier represents.

The second section contains a listing of the hidden and optional identifiers.

If an identifier is defined in a standard header file and it represents either a macro or a type, then the identifier is reserved if and only if that standard header file is explicitly included within the compilation unit. The programmer may define any of these reserved identifiers as long as it is in a different name space, but care should be taken not to compromise readability.

Note

Note that preprocessing directive keywords (for example, `include`), preprocessing operator names (for example, `defined`), and structure/union member names (for example, `tm_year`) are not included in this list because they pose no potential problems, since the compiler can unambiguously determine to which entity they refer, from the context in which they are used.

IDENTIFIER	HEADER	DESCRIPTION
__DATE__	N/A	Predefined macro (string literal)
__FILE__	N/A	Predefined macro (string literal)
__LINE__	N/A	Predefined macro (integral constant)
__TIME__	N/A	Predefined macro (string literal)
__STDC__	N/A	Predefined macro (integral constant)
_IOFBF	<stdio.h>	Macro (integral constant)
_IOLBF	<stdio.h>	Macro (integral constant)
_IONBF	<stdio.h>	Macro (integral constant)
BUFSIZ	<stdio.h>	Macro (integral constant)
CHAR_BIT	<limits.h>	Macro (preprocessor integral constant)
CHAR_MAX	<limits.h>	Macro (preprocessor integral constant)
CHAR_MIN	<limits.h>	Macro (preprocessor integral constant)
CLOCKS_PER_SEC	<time.h>	Macro (arithmetic expression)
DBL_DIG	<float.h>	Macro (integral expression)
DBL_EPSILON	<float.h>	Macro (`double` expression)
DBL_MANT_DIG	<float.h>	Macro (integral expression)
DBL_MAX	<float.h>	Macro (`double` expression)
DBL_MAX_10_EXP	<float.h>	Macro (integral expression)
DBL_MAX_EXP	<float.h>	Macro (integral expression)
DBL_MIN_EXP	<float.h>	Macro (integral expression)
EDOM	<errno.h>	Macro (integral constant)
EOF	<stdio.h>	Macro (negative integral constant)
ERANGE	<errno.h>	Macro (integral constant)
EXIT_FAILURE	<stdlib.h>	Macro (integral expression)
EXIT_SUCCESS	<stdlib.h>	Macro (integral expression)
FILE	<stdio.h>	Type
FILENAME_MAX	<stdio.h>	Macro (integral constant)
FLT_DIG	<float.h>	Macro (integral expression)
FLT_EPSILON	<float.h>	Macro (`float` expression)
FLT_MANT_DIG	<float.h>	Macro (integral expression)
FLT_MAX	<float.h>	Macro (`float` expression)
FLT_MAX_10_EXP	<float.h>	Macro (integral expression)
FLT_MAX_EXP	<float.h>	Macro (integral expression)

IDENTIFIER	HEADER	DESCRIPTION
FLT_RADIX	<float.h>	Macro (preprocessor integral constant)
FLT_ROUNDS	<float.h>	Macro (integral expression)
FOPEN_MAX	<stdio.h>	Macro (integral constant)
HUGE_VAL	<math.h>	Macro (double expression)
INT_MAX	<limits.h>	Macro (preprocessor integral constant)
INT_MIN	<limits.h>	Macro (preprocessor integral constant)
LC_ALL	<locale.h>	Macro (integral constant)
LC_COLLATE	<locale.h>	Macro (integral constant)
LC_CTYPE	<locale.h>	Macro (integral constant)
LC_MONETARY	<locale.h>	Macro (integral constant)
LC_NUMERIC	<locale.h>	Macro (integral constant)
LC_TIME	<locale.h>	Macro (integral constant)
LDBL_DIG	<float.h>	Macro (integral expression)
LDBL_EPSILON	<float.h>	Macro (long double expression)
LDBL_MANT_DIG	<float.h>	Macro (integral expression)
LDBL_MAX	<float.h>	Macro (long double expression)
LDBL_MAX_10_EXP	<float.h>	Macro (integral expression)
LDBL_MAX_EXP	<float.h>	Macro (integral expression)
LDBL_MIN	<float.h>	Macro (long double expression)
LDBL_MIN_10_EXP	<float.h>	Macro (integral expression)
LDBL_MIN_EXP	<float.h>	Macro (integral expression)
LONG_MAX	<limits.h>	Macro (preprocessor integral constant)
LONG_MIN	<limits.h>	Macro (preprocessor integral constant)
L_tmpnam	<stdio.h>	Macro (integral constant)
MB_CUR_MAX	<stdlib.h>	Macro (integral expression)
MB_LEN_MAX	<limits.h>	Macro (preprocessor integral constant)
NDEBUG	<assert.h>	Macro reference (user defined)
NULL	<locale.h>	Macro (null pointer constant)
NULL	<stddef.h>	Macro (null pointer constant)
NULL	<stdio.h>	Macro (null pointer constant)
NULL	<stdlib.h>	Macro (null pointer constant)
NULL	<string.h>	Macro (null pointer constant)
NULL	<time.h>	Macro (null pointer constant)

IDENTIFIER	HEADER	DESCRIPTION
RAND_MAX	<stdlib.h>	Macro (integral constant)
SCHAR_MAX	<limits.h>	Macro (preprocessor integral constant)
SCHAR_MIN	<limits.h>	Macro (preprocessor integral constant)
SEEK_CUR	<stdio.h>	Macro (integral constant)
SEEK_END	<stdio.h>	Macro (integral constant)
SEEK_SET	<stdio.h>	Macro (integral constant)
SHRT_MAX	<limits.h>	Macro (preprocessor integral constant)
SHRT_MIN	<limits.h>	Macro (preprocessor integral constant)
SIGABT	<signal.h>	Macro (positive integral constant)
SIGFPE	<signal.h>	Macro (positive integral constant)
SIGILL	<signal.h>	Macro (positive integral constant)
SIGNT	<signal.h>	Macro (positive integral constant)
SIGSEGV	<signal.h>	Macro (positive integral constant)
SIGTERM	<signal.h>	Macro (positive integral constant)
SIG_DFL	<signal.h>	Macro ("void (*)(int)" constant)
SIG_ERR	<signal.h>	Macro ("void (*)(int)" constant)
SIG_IGN	<signal.h>	Macro ("void (*)(int)" constant)
TMP_MAX	<stdio.h>	Macro (integral constant)
UCHAR_MAX	<limits.h>	Macro (preprocessor integral constant)
UINT_MAX	<limits.h>	Macro (preprocessor integral constant)
ULONG_MAX	<limits.h>	Macro (preprocessor integral constant)
USHRT_MAX	<limits.h>	Macro (preprocessor integral constant)
abort	<stdlib.h>	Function or macro function
abs	<stdlib.h>	Function or macro function
acos	<math.h>	Function or macro function

IDENTIFIER	HEADER	DESCRIPTION
asctime	<time.h>	Function or macro function
asin	<math.h>	Function or macro function
assert	<assert.h>	Macro function (void)
atan	<math.h>	Function or macro function
atan2	<math.h>	Function or macro function
atexit	<stdlib.h>	Function or macro function
atof	<stdlib.h>	Function or macro function
atoi	<stdlib.h>	Function or macro function
atol	<stdlib.h>	Function or macro function
auto	N/A	Keyword
break	N/A	Keyword
bsearch	<stdlib.h>	Function or macro function
calloc	<stdlib.h>	Function or macro function
case	N/A	Keyword
ceil	<math.h>	Function or macro function
char	N/A	Keyword
clearerr	<stdio.h>	Function or macro function
clock	<time.h>	Function or macro function
clock_t	<time.h>	Type (arithmetic)
const	N/A	Keyword
continue	N/A	Keyword
cos	<math.h>	Function or macro function
cosh	<math.h>	Function or macro function
ctime	<time.h>	Function or macro function
default	N/A	Keyword
difftime	<time.h>	Function or macro function
div	<stdlib.h>	Function or macro function
div_t	<stdlib.h>	Type (structure)
do	N/A	Keyword
double	N/A	Keyword
else	N/A	Keyword
enum	N/A	Keyword
errno	<errno.h>	Macro (int modifiable lvalue)
exit	<stdlib.h>	Function or macro function
exp	<math.h>	Function or macro function
extern	N/A	Keyword
fabs	<math.h>	Function or macro function
fclose	<stdio.h>	Function or macro function
feof	<stdio.h>	Function or macro function
ferror	<stdio.h>	Function or macro function
fflush	<stdio.h>	Function or macro function
fgetc	<stdio.h>	Function or macro function

IDENTIFIER	HEADER	DESCRIPTION
fgetpos	<stdio.h>	Function or macro function
fgets	<stdio.h>	Function or macro function
float	N/A	Keyword
floor	<math.h>	Function or macro function
fmod	<math.h>	Function or macro function
fopen	<stdio.h>	Function or macro function
for	N/A	Keyword
fpos_t	<stdio.h>	Type
fprintf	<stdio.h>	Function or macro function
fputc	<stdio.h>	Function or macro function
fputs	<stdio.h>	Function or macro function
fread	<stdio.h>	Function or macro function
free	<stdlib.h>	Function or macro function
freopen	<stdio.h>	Function or macro function
frexp	<math.h>	Function or macro function
fscanf	<stdio.h>	Function or macro function
fseek	<stdio.h>	Function or macro function
fsetpos	<stdio.h>	Function or macro function
ftell	<stdio.h>	Function or macro function
fwrite	<stdio.h>	Function or macro function
getc	<stdio.h>	Function or macro function
getchar	<stdio.h>	Function or macro function
getenv	<stdlib.h>	Function or macro function
gets	<stdio.h>	Function or macro function
gmtime	<time.h>	Function or macro function
goto	N/A	Keyword
if	N/A	Keyword
int	N/A	Keyword
isalnum	<ctype.h>	Function or macro function
isalpha	<ctype.h>	Function or macro function
iscntrl	<ctype.h>	Function or macro function
isdigit	<ctype.h>	Function or macro function
isgraph	<ctype.h>	Function or macro function
islower	<ctype.h>	Function or macro function
isprint	<ctype.h>	Function or macro function
ispunct	<ctype.h>	Function or macro function
isspace	<ctype.h>	Function or macro function
isupper	<ctype.h>	Function or macro function
isxdigit	<ctype.h>	Function or macro function
jmp_buf	<setjmp.h>	Type (array)
labs	<stdlib.h>	Function or macro function
lconv	<locale.h>	Structure tag

IDENTIFIER	HEADER	DESCRIPTION
ldexp	<math.h>	Function or macro function
ldiv	<stdlib.h>	Function or macro function
ldiv_t	<stdlib.h>	Type (structure)
localeconv	<locale.h>	Function or macro function
localtime	<time.h>	Function or macro function
log	<math.h>	Function or macro function
log10	<math.h>	Function or macro function
long	N/A	Keyword
longjmp	<setjmp.h>	Function or macro function
main	N/A	Function reference (user defined)
malloc	<stdlib.h>	Function or macro function
mblen	<stdlib.h>	Function or macro function
mbstowcs	<stdlib.h>	Function or macro function
mbtowc	<stdlib.h>	Function or macro function
memchr	<string.h>	Function or macro function
memcmp	<string.h>	Function or macro function
memcpy	<string.h>	Function or macro function
memmove	<string.h>	Function or macro function
memset	<string.h>	Function or macro function
mktime	<time.h>	Function or macro function
modf	<math.h>	Function or macro function
offsetof	<stddef.h>	Macro function (size_t)
perror	<stdio.h>	Function or macro function
pow	<math.h>	Function or macro function
printf	<stdio.h>	Function or macro function
ptrdiff_t	<stddef.h>	Type (signed integral)
putc	<stdio.h>	Function or macro function
putchar	<stdio.h>	Function or macro function
puts	<stdio.h>	Function or macro function
qsort	<stdlib.h>	Function or macro function
raise	<signal.h>	Function or macro function
rand	<stdlib.h>	Function or macro function
realloc	<stdlib.h>	Function or macro function
register	N/A	Keyword
remove	<stdio.h>	Function or macro function
rename	<stdio.h>	Function or macro function
return	N/A	Keyword
rewind	<stdio.h>	Function or macro function
scanf	<stdio.h>	Function or macro function
setbuf	<stdio.h>	Function or macro function
setjmp	<setjmp.h>	Function or macro function

IDENTIFIER	HEADER	DESCRIPTION
setlocale	<locale.h>	Function or macro function
setvbuf	<stdio.h>	Function or macro function
short	N/A	Keyword
sig_atomic_t	<signal.h>	Type (integral)
signal	<signal.h>	Function or macro function
signed	N/A	Keyword
sin	<math.h>	Function or macro function
sinh	<math.h>	Function or macro function
size_t	<stddef.h>	Type (unsigned integral)
size_t	<stdio.h>	Type (unsigned integral)
size_t	<stdlib.h>	Type (unsigned integral)
size_t	<string.h>	Type (unsigned integral)
size_t	<time.h>	Type (unsigned integral)
sizeof	N/A	Keyword
sprintf	<stdio.h>	Function or macro function
sqrt	<math.h>	Function or macro function
srand	<stdlib.h>	Function or macro function
sscanf	<stdio.h>	Function or macro function
static	N/A	Keyword
stderr	<stdio.h>	Macro ("FILE *" expression)
stdin	<stdio.h>	Macro ("FILE *" expression)
stdout	<stdio.h>	Macro ("FILE *" expression)
strcat	<string.h>	Function or macro function
strchr	<string.h>	Function or macro function
strcmp	<string.h>	Function or macro function
strcoll	<string.h>	Function or macro function
strcpy	<string.h>	Function or macro function
strcspn	<string.h>	Function or macro function
strerror	<string.h>	Function or macro function
strftime	<time.h>	Function or macro function
strlen	<string.h>	Function or macro function
strncat	<string.h>	Function or macro function
strncmp	<string.h>	Function or macro function
strncpy	<string.h>	Function or macro function
strpbrk	<string.h>	Function or macro function
strrchr	<string.h>	Function or macro function
strspn	<string.h>	Function or macro function
strstr	<string.h>	Function or macro function
strtod	<stdlib.h>	Function or macro function
strtok	<string.h>	Function or macro function
strtol	<stdlib.h>	Function or macro function
strtoul	<stdlib.h>	Function or macro function

IDENTIFIER	HEADER	DESCRIPTION
`struct`	N/A	Keyword
`strxfrm`	`<string.h>`	Function or macro function
`switch`	N/A	Keyword
`system`	`<stdlib.h>`	Function or macro function
`tan`	`<math.h>`	Function or macro function
`tanh`	`<math.h>`	Function or macro function
`time`	`<time.h>`	Function or macro function
`time_t`	`<time.h>`	Type (arithmetic)
`tm`	`<time.h>`	Structure tag
`tmpfile`	`<stdio.h>`	Function or macro function
`tmpnam`	`<stdio.h>`	Function or macro function
`tolower`	`<ctype.h>`	Function or macro function
`toupper`	`<ctype.h>`	Function or macro function
`typedef`	N/A	Keyword
`ungetc`	`<stdio.h>`	Function or macro function
`union`	N/A	Keyword
`unsigned`	N/A	Keyword
`va_arg`	`<stdarg.h>`	Macro function (parameter)
`va_end`	`<stdarg.h>`	Macro function (`void`)
`va_list`	`<stdarg.h>`	Type
`va_start`	`<stdarg.h>`	Macro function (`void`)
`vfprintf`	`<stdio.h>`	Function or macro function
`void`	N/A	Keyword
`volatile`	N/A	Keyword
`vprintf`	`<stdio.h>`	Function or macro function
`vsprintf`	`<stdio.h>`	Function or macro function
`wchar_t`	`<stddef.h>`	Type (integral)
`wcstombs`	`<stdlib.h>`	Function or macro function
`wctomb`	`<stdlib.h>`	Function or macro function
`while`	N/A	Keyword

This section contains another list of identifiers, which may be reserved either for internal use (hidden) within an implementation, or for additional (optional) functionality that an implementation may provide, or for possible future functionality that may be adopted by the ANSI C standard. The specification for these identifiers uses a UNIX-like regular expression notation; "[A-Z]*" means "zero or more uppercase letters"; "[a-z]*" means "zero or more lowercase letters"; and "[0-9]*" means "zero or more digits". For maximal portability and maintainability, the programmer should avoid using these identifiers.

IDENTIFIER	HEADER	DESCRIPTION
__*	N/A	Hidden macros
_[A-Z]*	N/A	Hidden macros
_*	N/A	Hidden external names
E[0-9]*	<errno.h>	Additional macros
E[A-Z]*	<errno.h>	Additional macros
LC_[A-Z]*	<locale.h>	Additional macros
SIG_*	<signal.h>	Additional macros
SIG[A-Z]*	<signal.h>	Additional macros
acosf	<math.h>	Future function or macro function
acosl	<math.h>	Future function or macro function
asinf	<math.h>	Future function or macro function
asinl	<math.h>	Future function or macro function
atanf	<math.h>	Future function or macro function
atanl	<math.h>	Future function or macro function
atan2f	<math.h>	Future function or macro function
atan2l	<math.h>	Future function or macro function
ceilf	<math.h>	Future function or macro function
ceill	<math.h>	Future function or macro function
cosf	<math.h>	Future function or macro function
cosl	<math.h>	Future function or macro function
coshf	<math.h>	Future function or macro function
coshl	<math.h>	Future function or macro function
expf	<math.h>	Future function or macro function
expl	<math.h>	Future function or macro function
fabsf	<math.h>	Future function or macro function
fabsl	<math.h>	Future function or macro function
floorf	<math.h>	Future function or macro function
floorl	<math.h>	Future function or macro function
fmodf	<math.h>	Future function or macro function
fmodl	<math.h>	Future function or macro function
frexpf	<math.h>	Future function or macro function

IDENTIFIER	HEADER	DESCRIPTION
`frexpl`	`<math.h>`	Future function or macro function
`is[a-z]*`	`<ctype.h>`	Future functions or macro functions
`ldexpf`	`<math.h>`	Future function or macro function
`ldexpl`	`<math.h>`	Future function or macro function
`logf`	`<math.h>`	Future function or macro function
`logl`	`<math.h>`	Future function or macro function
`log10f`	`<math.h>`	Future function or macro function
`log10l`	`<math.h>`	Future function or macro function
`mem[a-z]*`	`<string.h>`	Future functions or macro functions
`modff`	`<math.h>`	Future function or macro function
`modfl`	`<math.h>`	Future function or macro function
`powf`	`<math.h>`	Future function or macro function
`powl`	`<math.h>`	Future function or macro function
`sinf`	`<math.h>`	Future function or macro function
`sinl`	`<math.h>`	Future function or macro function
`sinhf`	`<math.h>`	Future function or macro function
`sinhl`	`<math.h>`	Future function or macro function
`sqrtf`	`<math.h>`	Future function or macro function
`str[a-z]*`	`<stdlib.h>`	Future functions or macro functions
`str[a-z]*`	`<string.h>`	Future functions or macro functions
`tanf`	`<math.h>`	Future function or macro function
`tanl`	`<math.h>`	Future function or macro function
`tanhf`	`<math.h>`	Future function or macro function
`tanhl`	`<math.h>`	Future function or macro function
`to[a-z]*`	`<ctype.h>`	Future functions or macro functions
`wcs[a-z]*`	`<string.h>`	Future functions or macro functions

Appendix C: ANSI C Implementation-Defined Behavior

This chapter lists ANSI C implementation-defined behavior and locale-specific behavior.

Implementation-Defined Behavior

The following is a list of all ANSI C implementation-defined behavior.

Implementation-defined behavior (which applies to a correct program construct and correct data) depends upon the characteristics of the implementation.

Every implementation of ANSI C is required to supply a description of each of these characteristics. See your *LPI-C User's Guide* for further information concerning specific behavior for your machine.

1. Translation

i. how a diagnostic is identified

ii. whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character that is implementation-defined

2. Environment

i. the semantics of the arguments to `main`

ii. what constitutes an interactive device

3. Identifiers

i. the number of significant initial characters in an identifier without external linkage (at least 31 guaranteed by any ANSI C implementation)

ii. the number of significant initial characters in an identifier with external linkage (at least 6 are guaranteed by any ANSI C implementation)

iii. whether case distinctions are significant in an identifier with external linkage

4. Characters

i. the members of the source and execution character sets, except as explicitly specified by the ANSI C Standard

ii. the shift states used for the encoding of multibyte characters

iii. the value of each escape sequence

iv. the number of bits in a character in the execution character set

v. the mapping of members of the source character set (in character constants and string literals) to members of the execution character set

vi. the value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant

vii. the value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character

viii. the current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant

ix. whether a "plain" `char` has the same range of values as `signed char` or `unsigned char`

5. Integers

i. the representations and sets of values of the various types of integers

ii. the result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented

iii. the sign of the remainder on integer division

iv. the results of bitwise operations on signed integers

v. the result of a right shift of a negative-valued signed integral type

6. Floating point

i. the representations and sets of values of the various types of floating-point numbers

ii. the direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value

iii. the direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number

7. Arrays and pointers

i. the type of integer required to hold the maximum size of an array; that is, the type of the `sizeof` operator (`size_t`)

ii. the result of casting a pointer to an integer or vice versa

iii. the type of integer required to hold the difference between two pointers to members of the same array (that is, `ptrdiff_t`)

8. Registers

i. the extent to which objects can actually be placed in registers by use of the `register` storage-class specifier

9. Structures, unions, enumerations, and bit-fields

i. a member of a union object is accessed using a member of a different type

ii. the padding and alignment of members of structures (this should present no problem unless binary data written by one implementation are read by another)

iii. whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field

iv. the order of allocation of bit-fields within an `int`

v. whether a bit-field can straddle a storage-unit boundary

vi. the integer type chosen to represent the values of an enumeration type

10. Type qualifiers

i. what constitutes an access to an object that has volatile-qualified type

11. Declarators

i. the maximum number of pointer, array, and function declarators that may modify an arithmetic, structure, or union type

12. Statements

i. the maximum number of `case` values in a `switch` statement

13. Preprocessing directives

i. whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set, and whether such a character constant may have a negative value

ii. the method for locating includable source files

iii. the support of quoted names for includable source files

iv. the mapping of source file character sequences

v. the behavior on each recognized `#pragma` directive

vi. the definitions for `__DATE__` and `__TIME__` when, respectively, the date and time of translation are not available

vii. the maximum nesting level for `#included` files

14. Library functions

i. the null pointer constant to which the macro **NULL** expands

ii. the diagnostic printed by, and the termination behavior of, the `assert` function

iii. the sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` functions

iv. the values returned by the mathematics functions on domain errors

v. whether the mathematics functions set the integer expression `errno` to the value of the macro **ERANGE** on underflow range errors

vi. whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero

vii. the set of signals for the `signal` function

viii. the semantics for each signal recognized by the `signal` function

ix. the default handling and the handling at program startup for each signal recognized by the `signal` function

x. if the equivalent of "`signal(sig, SIG_DFL);`" is not executed prior to the call of a signal handler, the blocking of the signal that is performed

xi. whether the default handling is reset if the `SIGILL` signal is received by a handler specified to the `signal` function

xii. whether the last line of a text stream requires a terminating new-line character

xiii. whether space characters that are written out to a text stream immediately before a new-line character appear when read in

xiv. the number of null characters that may be appended to data written to a binary stream

xv. whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file

xvi. whether a write on a text stream causes the associated file to be truncated beyond that point

xvii. the characteristics of file buffering

xviii. whether a zero-length file actually exists

xix. the rules for composing valid file names

xx. whether the same file can be open multiple times

xxi. the effect of the `remove` function on an open file

xxii. the effect if a file with the new name exists prior to a call to the `rename` function

xxiii. the output for `%p` conversion in the `fprintf` function

xxiv. the input for `%p` conversion in the `fscanf` function

xxv. the interpretation of a hyphen (-) character that is neither the first nor the last character in the scanlist for `%[` conversion in the `fscanf` function

xxvi. the value to which the macro `errno` is set by the `fgetpos` or `ftell` function on failure

xxvii. the messages generated by the `perror` function

xxviii. the behavior of the `abort` function with regard to open and temporary files

xxix. the status returned by the `exit` function if the value of the argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE`

xxx. the set of environment names and the method for altering the environment list used by the `getenv` function

xxxi. the contents and mode of execution of the string by the `system` function

xxxii. the contents of the error message strings returned by the `strerror` function

xxxiii. the local time zone and Daylight Saving Time

xxxiv. the era for the `clock` function

Locale-specific behavior

The following characteristics of a hosted environment are locale-specific:

- the content of the execution character set, in addition to the required members
- the direction of printing
- the decimal-point character
- the implementation-defined aspects of character testing and case mapping functions
- the collation sequence of the execution character set
- the formats of time and date

Appendix D: Compilation and Numerical Limits

This chapter lists the compilation and numerical limits of LPI-C.

Compilation Limits

The following lists the minimum compilation limits which the ANSI C Standard imposes on implementation. The values enclosed in < > correspond to the maximum LPI-C limit.

Note
"No limit" implies no specific compiler limit, but may be limited or restricted by system resources, etc.

- 15 nesting levels of compound statements, iteration control structures, and selection control structures < 100 >
- 8 nesting levels of conditional inclusion < No limit >
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration < 15 >
- 31 nesting levels of parenthesized declarators within a full declarator < No limit >
- 32 nesting levels of parenthesized expressions within a full expression < 100 >
- 31 significant initial characters in an internal identifier or a macro name
- 6 significant initial characters in an external identifier < System dependent >
- 511 external identifiers in one translation unit < No limit >
- 127 identifiers with block scope declared in one block < No limit >
- 1024 macro identifiers simultaneously defined in one translation unit < No limit >

- 31 parameters in one function definition < No limit >
- 31 arguments in one function call < 63 >
- 31 parameters in one macro definition < 256 >
- 31 arguments in one macro invocation < 256 >
- 509 characters in a logical source line < No limit >
- 509 characters in a character string literal or wide string literal (after concatenation) < 32767 >
- 32767 bytes in an object < 2147483647 >
- 8 nesting levels for `#included` files < No limit >
- 257 case labels for a switch statement (excluding those for any nested switch statements) < Unlimited >
- 127 members in a single structure or union < No limit >
- 127 enumeration constants in a single enumeration < No limit >
- 15 levels of nested structure or union definitions in a single struct-declaration-list < No limit >

Numerical Limits

This section lists the minimum numerical limits of the values specified in the headers `<limits.h>` and `<float.h>`.

Sizes of Integral Types <limits.h>

The values given below are replaced by constant expressions suitable for use in `#if` preprocessing directives. Morever, except for `CHAR_BIT` and `MB_LEN_MAX`, the following are replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integral promotions. Their values are equal or greater in magnitude (absolute value) to those shown, with the same sign.

CONSTANT	VALUE	DESCRIPTION
`CHAR_BIT`	8	maximum number of bits for smallest object that is not a bit-field (byte)

CONSTANT	VALUE	DESCRIPTION
SCHAR_MIN	−127	minimum value for an object of type signed `char`
SCHAR_MAX	+127	maximum value for an object of type signed `char`
UCHAR_MAX	255	maximum value for an object of type unsigned `char`
CHAR_MIN	−127	minimum value for an object of type `char`
CHAR_MAX	+127	maximum value for an object of type `char`
MB_LEN_MAX	1	maximum number of bytes in a multibyte character, for any supported locale
SHRT_MIN	−32767	minimum value for an object of type `short int`
SHRT_MAX	+32767	maximum value for an object of type `short int`
USHRT_MAX	65535	maximum value for an object of type unsigned `short int`
INT_MIN	−32767	minimum value for an object of type `int`
INT_MAX	+32767	maximum value for an object of type `int`
UINT_MAX	65535	maximum value for an object of type `unsigned int`
LONG_MIN	−2147483647	minimum value for an object of type `long int`

CONSTANT	VALUE	DESCRIPTION
LONG_MAX	+2147483647	maximum value for an object of type `long int`
ULONG_MAX	4294967295	maximum value for an object of type `unsigned long int`

Characteristics of Floating Types <float.h>

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about the implementation's floating-point arithmetic.

The following parameters are used to define the model for each floating-point type:

- s sign $(+ -1)$
- b base or radix of exponent representation (an integer > 1)
- e exponent (an integer between a minimum $e_{\min}$ and a maximum $e_{\max}$)
- p precision (the number of base-b digits in the significand)
- f_k nonnegative integers less than b (the significand digits)

A normalized floating-point number ($f_1 > 0$ if $\neq 0$) is defined by the following model:

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \quad e_{\min} \leq e \leq e_{\max}$$

Note
This model precludes floating-point representations other than sign-magnitude.

Of the values in the `<float.h>` header, FLT_RADIX is a constant expression suitable for use in `#if` preprocessing directives; all other values need not be constant expressions. All except FLT_RADIX and FLT_ROUNDS have separate names for all three floating-point types. The floating-point model representation is provided for all values except FLT_ROUNDS.

The rounding mode for floating-point addition is characterized by the value of FLT_ROUNDS:

−1	indeterminable
0	toward zero
1	to nearest
2	toward positive infinity
3	toward negative infinity

All other values for FLT_ROUNDS characterize implementation-defined rounding behavior.

The values given in the following list are replaced by implementation-defined expressions that are equal or greater in magnitude (absolute value) to those shown, with the same sign.

- radix of exponent representation, b

  FLT_RADIX 2

- number of base- FLT_RADIX digits in the floating-point significand, p

  FLT_MANT_DIG
  DBL_MANT_DIG
  LDBL_MANT_DIG

- number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits,

$$\lfloor (p - 1) \times \log_{10} b \rfloor + \begin{cases} 1 & \text{if } b \text{ is a power of 10} \\ 0 & \text{otherwise} \end{cases}$$

  FLT_DIG 6

  DBL_DIG 10

  LDBL_DIG 10

- minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number, e_{min}

  FLT_MIN_EXP
  DBL_MIN_EXP
  LDBL_MIN_EXP

- minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil \log_{10} b^{e_{min}-1} \rceil$

FLT_MIN_10_EXP	−37
DBL_MIN_10_EXP	−37
LDBL_MIN_10_EXP	−37

- maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number, e_{max}

  FLT_MAX_EXP
  DBL_MAX_EXP
  LDBL_MAX_EXP

- maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers,

  $\lfloor \log_{10}((1 - b^{-p}) \times b^{e_{max}}) \rfloor$

FLT_MAX_10_EXP	+37
DBL_MAX_10_EXP	+37
LDBL_MAX_10_EXP	+37

The values given in the following list are replaced by implementation-defined expressions with values that are equal to or greater than those shown.

- maximum representable finite floating-point number, $(1 - b^{-p}) \times b^{e_{max}}$

FLT_MAX	1E+37
DBL_MAX	1E+37
LDBL_MAX	1E+37

The values given in the following list are replaced by implementation-defined expressions with values that are equal to or less than those shown.

- the difference between 1.0 and the least value greater than 1.0 that is representable in the given floating pont type, b^{1-p}

FLT_EPSILON	1E-5
DBL_EPSILON	1E-9
LDBL_EPSILON	1E-9

- minimum normalized positive floating-point number, $b^{e_{min}-1}$

FLT_MIN	1E-37
DBL_MIN	1E-37
LDBL_MIN	1E-37

Examples

The following describes an artificial floating-point representation that meets the minimum requirements of the Standard, and the appropriate values in a `<float.h>` header for type `float`:

$$x = s \times 16^e \times \sum_{k=1}^{6} f_k \times 16^{-k}, \quad -31 \leq e \leq +32$$

FLT_RADIX	16
FLT_MANT_DIG	6
FLT_EPSILON	9.53674316E-07F
FLT_DIG	6
FLT_MIN_EXP	-31
FLT_MIN	2.93873588E-39F
FLT_MIN_10_EXP	-38
FLT_MAX_EXP	+32
FLT_MAX	3.40282347E+38F
FLT_MAX_10_EXP	+38

The following describes floating-point representations that also meet the requirements for single-precision and double-precision normalized numbers in the "IEEE Standard for Binary Floating-Point Arithmetic" (ANSI/IEEE Std 754-1985), and the appropriate values in a <float.h> header for types float and double :

$$x_f = s \times 2^e \times \sum_{k=1}^{24} f_k \times 2^{-k}, \quad -125 \leq e \leq +128$$

$$x_d = s \times 2^e \times \sum_{k=1}^{53} f_k \times 2^{-k}, \quad -1021 \leq e \leq +1024$$

LT_RADIX	2
FLT_MANT_DIG	24
FLT_EPSILON	1.19209290E-07F
FLT_DIG	6
FLT_MIN_EXP	-125
FLT_MIN	1.17549435E-38F
FLT_MIN_10_EXP	-37
FLT_MAX_EXP	+128
FLT_MAX	3.40282347E+38F
FLT_MAX_10_EXP	+38
DBL_MANT_DIG	53
DBL_EPSILON	2.2204460492503131E-16
DBL_DIG	15
DBL_MIN_EXP	-1021
DBL_MIN	2.2250738585072016E-308
DBL_MIN_10_EXP	-307
DBL_MAX_EXP	+1024
DBL_MAX	1.7976931348623157E+308
DBL_MAX_10_EXP	+308

Note
The floating-point model in that standard sums powers of *b* from zero, so the values of the exponent limits are one less than shown here.

The values shown above for FLT_EPSILON and DBL_EPSILON are appropriate for the ANSI/IEEE Std 754-1985 default rounding mode (to nearest). Their values may differ for other rounding modes. See Chapter 4 for information concerning conditional inclusion.

Appendix E: Unspecified and Undefined Behavior

This chapter lists unspecified and undefined behavior.

Unspecified Behavior

Unspecified behavior applies to a correct construct and correct data for which the Standard imposes no requirements.

The following behavior is unspecified:

- the manner and timing of static initialization
- the behavior if a printable character is written when the active position is at the final position of a line
- the behavior if a backspace character is written when the active position is at the initial position of a line
- the behavior if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position
- the behavior if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position
- the representations of floating types
- the order in which expressions are evaluated in any order conforming to the precedence rules, even in the presence of parentheses
- the order in which side effects take place
- the order in which the function designator and the arguments in a function call are evaluated
- the alignment of the addressable storage unit allocated to hold a bit-field
- the layout of storage for parameters
- the order in which # and ## operations are evaluated during macro substitution

- whether `errno` is a macro or an external identifier
- whether `setjmp` is a macro or an external identifier
- whether `va_end` is a macro or an external identifier
- the value of the file position indicator after a successful call to the `ungetc` function for a text stream, until all pushed-back characters are read or discarded
- the details of the value stored by the `fgetpos` function on success
- the details of the value returned by the `ftell` function for a text stream on success
- the order and contiguity of storage allocated by the `calloc`, `malloc`, and `realloc` functions
- which of two members that compare as equal is returned by the `bsearch` function
- the order in an array sorted by the `qsort` function of two members that compare as equal
- the encoding of the calendar time returned by the `time` function

Undefined Behavior

Undefined behavior applies to erroneous and/or nonportable program constructs for which the Standard imposes no requirements.

The following behavior is undefined:

- a nonempty source file does not end in a new-line character, ends in new-line character immediately preceded by a backslash character, or ends in a partial preprocessing token or comment
- a character not in the required character set is encountered in a source file, except in a preprocessing token that is never converted to a token, a character constant, a string literal, or a comment
- a comment, string literal, character constant, or header name contains an invalid multibyte character or does not begin and end in the initial shift state
- an unmatched ' or " character is encountered on a logical source line during tokenization
- the same identifier is used more than once as a label in the same function

- an identifier is used that is not visible in the current scope
- identifiers that are intended to denote the same entity differ in a character beyond the minimal significant characters
- the same identifier has both internal and external linkage in the same translation unit
- an identifier with external linkage is used but there does not exist exactly one external definition in the program for the identifier
- the value stored in a pointer that referred to an object with automatic storage duration is used
- two declarations of the same object or function specify types that are not compatible
- an unspecified escape sequence is encountered in a character constant or a string literal
- an attempt is made to modify a string literal of either form
- a character string literal token is adjacent to a wide string literal token
- the characters '/, , ", or /* are encountered between the < and > delimiters or the characters ', \, or /* are encountered between the " delimiters in the two forms of a header name preprocessing token
- an arithmetic conversion produces a result that cannot be represented in the space provided
- an lvalue with an incomplete type is used in a context that requires the value of the designated object
- the value of a `void` expression is used or an implicit conversion (except to `void`) is applied to a `void` expression
- an object is modified more than once, or is modified and accessed other than to determine the new value, between two sequence points
- an arithmetic operation is invalid (such as division or modulus by 0) or produces a result that cannot be represented in the space provided (such as overflow or underflow)
- an object has its stored value accessed by an lvalue that does not have one of the following types: the declared type of the object, a qualified version of the declared type of the object, the signed or unsigned type corresponding to the declared type of the object, the signed or unsigned type corresponding to a qualified version of the declared type of the object, an aggregate or union type that

(recursively) includes one of the aforementioned types among its members, or a character type

- an argument to a function is a `void` expression
- for a function call without a function prototype, the number of arguments does not agree with the number of parameters
- for a function call without a function prototype, if the function is defined without a function prototype, and the types of the arguments after promotion do not agree with those of the parameters after promotion
- if a function is called with a function prototype and the function is not defined with a compatible type
- a function that accepts a variable number of arguments is called without a function prototype that ends with an ellipsis
- an invalid array reference, null pointer reference, or reference to an object declared with automatic storage duration in a terminated block occurs
- a pointer to a function is converted to point to a function of a different type and used to call a function of a type not compatible with the original type
- a pointer to a function is converted to a pointer to an object or a pointer to an object is converted to a pointer to a function
- a pointer is converted to other than an integral or pointer type
- a pointer that is not to a member of an array object is added to or subtracted from
- pointers that are not to the same array object are subtracted
- an expression is shifted by a negative number or by an amount greater than or equal to the width in bits of the expression being shifted
- pointers are compared using a relational operator that do not point to the same aggregate or union
- an object is assigned to an overlapping object
- an identifier for an object is declared with no linkage and the type of the object is incomplete after its declarator, or after its init-declarator if it has an initializer

- a function is declared at block scope with a storage-class specifier other than extern
- a bit-field is declared with a type other than int, signed int, or unsigned int
- an attempt is made to modify an object with const-qualified type by means of an lvalue with non-const-qualified type
- an attempt is made to refer to an object with volatile-qualified type by means of an lvalue with non-volatile-qualified type
- the value of an uninitialised object that has automatic storage duration is used before a value is assigned
- an object with aggregate or union type with static storage duration has a non-brace-enclosed initializer, or an object with aggregate or union type with automatic storage duration has either a single expression initializer with a type other than that of the object or a non-brace-enclosed initializer
- the value of a function is used, but no value was returned
- a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation
- an identifier for an object with internal linkage and an incomplete type is declared with a tentative definition
- the token defined is generated during the expansion of a `#if` or `#elif` preprocessing directive
- the `#include` preprocessing directive that results after expansion does not match one of the two header name forms
- a macro argument consists of no preprocessing tokens
- there are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directive lines
- the result of the preprocessing concatenation operator `##` is not a valid preprocessing token
- the `#line` preprocessing directive that results after expansion does not match one of the two well-defined forms
- one of the following identifiers is the subject of a `#define` or `#undef` preprocessing directive: defined, `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, or `__STDC__`

- an attempt is made to copy an object to an overlapping object by use of a library function other than memmove
- the effect if the program redefines a reserved external identifier
- the effect if a standard header is included within an external definition; is included for the first time after the first reference to any of the functions or objects it declares, or to any of the types or macros it defines; or is included while a macro is defined with a name the same as a keyword
- a macro definition of `errno` is suppressed to obtain access to an actual object
- the parameter member-designator of an `offsetof` macro is an invalid right operand of the operator for the type parameter or designates bit-field member of a structure
- a library function argument has an invalid value, unless the behavior is specified explicitly
- a library function that accepts a variable number of arguments is not declared
- the macro definition of `assert` is suppressed to obtain access to an actual function
- the argument to a character handling function is out of the domain
- a macro definition of `setjmp` is suppressed to obtain access to an actual function
- an invocation of the `setjmp` macro occurs in a context other than as the controlling expression in a selection or iteration statement, or in a comparison with an integral constant expression (possibly as implied by the unary ! operator) as the controlling expression of a selection or iteration statement, or as an expression statement (possibly cast to `void`)
- an object of automatic storage class that does not have volatile-qualified type has been changed between a `setjmp` invocation and a `longjmp` call and then has its value accessed
- the `longjmp` function is invoked from a nested signal routine

- a signal occurs other than as the result of calling the `abort` or `raise` function, and the signal handler calls any function in the standard library other than the signal function itself or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type volatile `sig_atomic_t`
- the value of `errno` is referred to after a signal occurs other than as the result of calling the `abort` or `raise` function and the corresponding signal handler calls the signal function such that it returns the value `SIG_ERR`
- the macro `va_arg` is invoked with the parameter `ap` that was passed to a function that invoked the macro `va_arg` with the same parameter
- a macro definition of `va_start`, `va_arg`, or `va_end` or a combination thereof is suppressed to obtain access to an actual function
- the parameter *parmN* of a `va_start` macro is declared with the register storage class, or with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions
- there is no actual next argument for a `va_arg` macro invocation
- the type of the actual next argument in a variable argument list disagrees with the type specified by the `va_arg` macro
- the `va_end` macro is invoked without a corresponding invocation of the `va_start` macro
- a return occurs from a function with a variable argument list initialized by the `va_start` macro before the `va_end` macro is invoked
- the stream for the `fflush` function points to an input stream or to an update stream in which the most recent operation was input
- an output operation on an update stream is followed by an input operation without an intervening call to the `fflush` function or a file positioning function, or an input operation on an update stream is followed by an output operation without an intervening call to a file positioning function
- the format for the `fprintf` or `fscanf` function does not match the argument list

- an invalid conversion specification is found in the format for the `fprintf` or `fscanf` function
- a `%%` conversion specification for the `fprintf` or `fscanf` function contains characters between the pair of `%` characters
- a conversion specification for the `fprintf` function contains an h or l with a conversion specifier other than `d`, `i`, `n`, `o`, `u`, `x`, or `X`, or an L with a conversion specifier other than `e`, `E`, `f`, `g`, or `G`
- a conversion specification for the `fprintf` function contains a `#` flag with a conversion specifier other than `o`, `x`, `X`, `e`, `E`, `f`, `g`, or `G`
- a conversion specification for the `fprintf` function contains a 0 flag with a conversion specifier other than `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, or `G`
- an aggregate or union, or a pointer to an aggregate or union, is an argument to the `fprintf` function, except for the conversion specifiers `%s` (for an array of character type) or `%p` (for a pointer to `void`)
- a single conversion by the `fprintf` function produces more than 509 characters of output
- a conversion specification for the `fscanf` function contains an h or l with a conversion specifier other than d, i, n, o, u, or x, or an L with a conversion specifier other than e, f, or g
- a pointer value printed by `%p` conversion by the `fprintf` function during a previous program execution is the argument for `%p` conversion by the `fscanf` function
- the result of a conversion by the `fscanf` function cannot be represented in the space provided, or the receiving object does not have an appropriate type
- the result of converting a string to a number by the `atof`, `atoi`, or `atol` function cannot be represented
- the value of a pointer that refers to space deallocated by a call to the `free` or `realloc` function is referred to
- the pointer argument to the `free` or `realloc` function does not match a pointer earlier returned by `calloc`, `malloc`, or `realloc`, or the object pointed to has been deallocated by a call to `free` or `realloc`

- a program executes more than one call to the `exit` function
- the result of an integer arithmetic function (`abs`, `div`, `labs`, or `ldiv`) cannot be represented
- the shift states for the `mblen`, `mbtowc`, and `wctomb` functions are not explicitly reset to the initial state when the `LC_CTYPE` category of the current locale is changed
- an array written to by a copying or concatenation function is too small
- an invalid conversion specification is found in the format for the `strftime` function

Index

M

INTERACTIVE

A Kodak Company

DOC0113-2Z