array type, the types, if otherwise identical, are also taken to agree. Finally, if one type specifies an old-style function, and the other an otherwise identical new-style function, with parameter declarations, the types are taken to agree.

If the first external declaration for a function or object includes the `static` specifier, the identifier has *internal linkage*; otherwise it has *external linkage*. Linkage is discussed in §A11.2.

An external declaration for an object is a definition if it has an initializer. An external object declaration that does not have an initializer, and does not contain the `extern` specifier, is a *tentative definition*. If a definition for an object appears in a translation unit, any tentative definitions are treated merely as redundant declarations. If no definition for the object appears in the translation unit, all its tentative definitions become a single definition with initializer 0.

Each object must have exactly one definition. For objects with internal linkage, this rule applies separately to each translation unit, because internally-linked objects are unique to a translation unit. For objects with external linkage, it applies to the entire program.

> Although the one-definition rule is formulated somewhat differently in the first edition of this book, it is in effect identical to the one stated here. Some implementations relax it by generalizing the notion of tentative definition. In the alternate formulation, which is usual in UNIX systems and recognized as a common extension by the Standard, all the tentative definitions for an externally-linked object, throughout all the translation units of a program, are considered together instead of in each translation unit separately. If a definition occurs somewhere in the program, then the tentative definitions become merely declarations, but if no definition appears, then all its tentative definitions become a definition with initializer 0.

A11. Scope and Linkage

A program need not all be compiled at one time: the source text may be kept in several files containing translation units, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, the *lexical scope* of an identifier, which is the region of the program text within which the identifier's characteristics are understood; and second, the scope associated with objects and functions with external linkage, which determines the connections between identifiers in separately compiled translation units.

A11.1 Lexical Scope

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces. These classes are: objects, functions, typedef names, and enum constants; labels; tags of structures, unions, and enumerations; and members of each structure or union individually.

> These rules differ in several ways from those described in the first edition of this manual. Labels did not previously have their own name space; tags of structures and unions each had a separate space, and in some implementations

enumeration tags did as well; putting different kinds of tags into the same space is a new restriction. The most important departure from the first edition is that each structure or union creates a separate name space for its members, so that the same name may appear in several different structures. This rule has been common practice for several years.

The lexical scope of an object or function identifier in an external declaration begins at the end of its declarator and persists to the end of the translation unit in which it appears. The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block. The scope of a label is the whole of the function in which it appears. The scope of a structure, union, or enumeration tag, or an enumeration constant, begins at its appearance in a type specifier, and persists to the end of the translation unit (for declarations at the external level) or to the end of the block (for declarations within a function).

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

A11.2 Linkage

Within a translation unit, all declarations of the same object or function identifier with internal linkage refer to the same thing, and the object or function is unique to that translation unit. All declarations for the same object or function identifier with external linkage refer to the same thing, and the object or function is shared by the entire program.

As discussed in §A10.2, the first external declaration for an identifier gives the identifier internal linkage if the `static` specifier is used, external linkage otherwise. If a declaration for an identifier within a block does not include the `extern` specifier, then the identifier has no linkage and is unique to the function. If it does include `extern`, and an external declaration for the identifier is active in the scope surrounding the block, then the identifier has the same linkage as the external declaration, and refers to the same object or function; but if no external declaration is visible, its linkage is external.

A12. Preprocessing

A preprocessor performs macro substitution, conditional compilation, and inclusion of named files. Lines beginning with `#`, perhaps preceded by white space, communicate with this preprocessor. The syntax of these lines is independent of the rest of the language; they may appear anywhere and have effect that lasts (independent of scope) until the end of the translation unit. Line boundaries are significant; each line is analyzed individually (but see §A12.2 for how to adjoin lines). To the preprocessor, a token is any language token, or a character sequence giving a file name as in the `#include` directive (§A12.4); in addition, any character not otherwise defined is taken as a token. However, the effect of white space characters other than space and horizontal tab is undefined within preprocessor lines.

Preprocessing itself takes place in several logically successive phases that may, in a particular implementation, be condensed.

1. First, trigraph sequences as described in §A12.1 are replaced by their equivalents. Should the operating system environment require it, newline characters are introduced between the lines of the source file.
2. Each occurrence of a backslash character \ followed by a newline is deleted, thus splicing lines (§A12.2).
3. The program is split into tokens separated by white-space characters; comments are replaced by a single space. Then preprocessing directives are obeyed, and macros (§§A12.3-A12.10) are expanded.
4. Escape sequences in character constants and string literals (§§A2.5.2, A2.6) are replaced by their equivalents; then adjacent string literals are concatenated.
5. The result is translated, then linked together with other programs and libraries, by collecting the necessary programs and data, and connecting external function and object references to their definitions.

A12.1 Trigraph Sequences

The character set of C source programs is contained within seven-bit ASCII, but is a superset of the ISO 646-1983 Invariant Code Set. In order to enable programs to be represented in the reduced set, all occurrences of the following trigraph sequences are replaced by the corresponding single character. This replacement occurs before any other processing.

`??=`	`#`	`??(`	`[`	`??<`	`{`
`??/`	`\`	`??)`	`]`	`??>`	`}`
`??'`	`^`	`??!`	`¦`	`??-`	`~`

No other such replacements occur.

Trigraph sequences are new with the ANSI standard.

A12.2 Line Splicing

Lines that end with the backslash character \ are folded by deleting the backslash and the following newline character. This occurs before division into tokens.

A12.3 Macro Definition and Expansion

A control line of the form

`# define` *identifier token-sequence*

causes the preprocessor to replace subsequent instances of the identifier with the given sequence of tokens; leading and trailing white space around the token sequence is discarded. A second `#define` for the same identifier is erroneous unless the second token sequence is identical to the first, where all white space separations are taken to be equivalent.

A line of the form

`# define` *identifier*`(` *identifier-list* `)` *token-sequence*

where there is no space between the first identifier and the `(`, is a macro definition with parameters given by the identifier list. As with the first form, leading and trailing white space around the token sequence is discarded, and the macro may be redefined only with a definition in which the number and spelling of parameters, and the token sequence, is identical.

A control line of the form

`# undef` *identifier*

causes the identifier's preprocessor definition to be forgotten. It is not erroneous to apply `#undef` to an unknown identifier.

When a macro has been defined in the second form, subsequent textual instances of the macro identifier followed by optional white space, and then by `(`, a sequence of tokens separated by commas, and a `)` constitute a call of the macro. The arguments of the call are the comma-separated token sequences; commas that are quoted or protected by nested parentheses do not separate arguments. During collection, arguments are not macro-expanded. The number of arguments in the call must match the number of parameters in the definition. After the arguments are isolated, leading and trailing white space is removed from them. Then the token sequence resulting from each argument is substituted for each unquoted occurrence of the corresponding parameter's identifier in the replacement token sequence of the macro. Unless the parameter in the replacement sequence is preceded by `#`, or preceded or followed by `##`, the argument tokens are examined for macro calls, and expanded as necessary, just before insertion.

Two special operators influence the replacement process. First, if an occurrence of a parameter in the replacement token sequence is immediately preceded by `#`, string quotes (`"`) are placed around the corresponding parameter, and then both the `#` and the parameter identifier are replaced by the quoted argument. A `\` character is inserted before each `"` or `\` character that appears surrounding, or inside, a string literal or character constant in the argument.

Second, if the definition token sequence for either kind of macro contains a `##` operator, then just after replacement of the parameters, each `##` is deleted, together with any white space on either side, so as to concatenate the adjacent tokens and form a new token. The effect is undefined if invalid tokens are produced, or if the result depends on the order of processing of the `##` operators. Also, `##` may not appear at the beginning or end of a replacement token sequence.

In both kinds of macro, the replacement token sequence is repeatedly rescanned for more defined identifiers. However, once a given identifier has been replaced in a given expansion, it is not replaced if it turns up again during rescanning; instead it is left unchanged.

Even if the final value of a macro expansion begins with `#`, it is not taken to be a preprocessing directive.

> The details of the macro-expansion process are described more precisely in the ANSI standard than in the first edition. The most important change is the addition of the `#` and `##` operators, which make quotation and concatenation admissible. Some of the new rules, especially those involving concatenation, are bizarre. (See example below.)

For example, this facility may be used for "manifest constants," as in

```
#define TABSIZE 100
int table[TABSIZE];
```

The definition

```
#define ABSDIFF(a, b)  ((a)>(b) ? (a)-(b) : (b)-(a))
```

defines a macro to return the absolute value of the difference between its arguments. Unlike a function to do the same thing, the arguments and returned value may have any arithmetic type or even be pointers. Also, the arguments, which might have side effects, are evaluated twice, once for the test and once to produce the value.

Given the definition

```
#define tempfile(dir)    #dir "/%s"
```

the macro call `tempfile(/usr/tmp)` yields

```
"/usr/tmp" "/%s"
```

which will subsequently be catenated into a single string. After

```
#define cat(x, y)    x ## y
```

the call `cat(var,123)` yields `var123`. However, the call `cat(cat(1,2),3))` is undefined: the presence of `##` prevents the arguments of the outer call from being expanded. Thus it produces the token string

```
cat  (  1  ,  2  )3
```

and `)3` (the catenation of the last token of the first argument with the first token of the second) is not a legal token. If a second level of macro definition is introduced,

```
#define xcat(x,y)    cat(x,y)
```

things work more smoothly; `xcat(xcat(1, 2), 3)` does produce `123`, because the expansion of `xcat` itself does not involve the `##` operator.

Likewise, `ABSDIFF(ABSDIFF(a,b),c)` produces the expected, fully-expanded result.

A12.4 File Inclusion

A control line of the form

`# include` *<filename>*

causes the replacement of that line by the entire contents of the file *filename*. The characters in the name *filename* must not include > or newline, and the effect is undefined if it contains any of ", ', \, or /*. The named file is searched for in a sequence of implementation-dependent places.

Similarly, a control line of the form

`# include` "*filename*"

searches first in association with the original source file (a deliberately implementation-dependent phrase), and if that search fails, then as if in the first form. The effect of using ', \, or /* in the filename remains undefined, but > is permitted.

Finally, a directive of the form

`# include` *token-sequence*

not matching one of the previous forms is interpreted by expanding the token sequence as for normal text; one of the two forms with <...> or "..." must result, and it is then treated as previously described.

`#include` files may be nested.

A12.5 Conditional Compilation

Parts of a program may be compiled conditionally, according to the following schematic syntax.

preprocessor-conditional:
    if-line text elif-parts else-part$_{opt}$ `#endif`

if-line:
    `# if` *constant-expression*
    `# ifdef` *identifier*
    `# ifndef` *identifier*

elif-parts:
    elif-line text
    elif-parts$_{opt}$

elif-line:
    `# elif` *constant-expression*

else-part:
    else-line text

else-line:
    `# else`

Each of the directives (if-line, elif-line, else-line, and `#endif`) appears alone on a line. The constant expressions in `#if` and subsequent `#elif` lines are evaluated in order until an expression with a non-zero value is found; text following a line with a zero value is discarded. The text following the successful directive line is treated normally. "Text" here refers to any material, including preprocessor lines, that is not part of the conditional structure; it may be empty. Once a successful `#if` or `#elif` line has been found and its text processed, succeeding `#elif` and `#else` lines, together with their text, are discarded. If all the expressions are zero, and there is an `#else`, the text following the `#else` is treated normally. Text controlled by inactive arms of the conditional is ignored except for checking the nesting of conditionals.

The constant expression in `#if` and `#elif` is subject to ordinary macro replacement. Moreover, any expressions of the form

    `defined` *identifier*

or

    `defined (` *identifier* `)`

are replaced, before scanning for macros, by `1L` if the identifier is defined in the preprocessor, and by `0L` if not. Any identifiers remaining after macro expansion are replaced by `0L`. Finally, each integer constant is considered to be suffixed with `L`, so that all arithmetic is taken to be long or unsigned long.

The resulting constant expression (§A7.19) is restricted: it must be integral, and may not contain `sizeof`, a cast, or an enumeration constant.

The control lines

    `#ifdef` *identifier*
    `#ifndef` *identifier*

are equivalent to

    `# if defined` *identifier*
    `# if ! defined` *identifier*

respectively.

> `#elif` is new since the first edition, although it has been available in some preprocessors. The `defined` preprocessor operator is also new.

A12.6 Line Control

For the benefit of other preprocessors that generate C programs, a line in one of the forms

`line` *constant* "*filename*"
`line` *constant*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the decimal integer constant and the current input file is named by the identifier. If the quoted filename is absent, the remembered name does not change. Macros in the line are expanded before it is interpreted.

A12.7 Error Generation

A preprocessor line of the form

`error` *token-sequence*$_{opt}$

causes the processor to write a diagnostic message that includes the token sequence.

A12.8 Pragmas

A control line of the form

`pragma` *token-sequence*$_{opt}$

causes the processor to perform an implementation-dependent action. An unrecognized pragma is ignored.

A12.9 Null Directive

A preprocessor line of the form

#

has no effect.

A12.10 Predefined Names

Several identifiers are predefined, and expand to produce special information. They, and also the preprocessor expression operator `defined`, may not be undefined or redefined.

`__LINE__` A decimal constant containing the current source line number.

`__FILE__` A string literal containing the name of the file being compiled.

`__DATE__` A string literal containing the date of compilation, in the form `"Mmm dd yyyy"`.

`__TIME__` A string literal containing the time of compilation, in the form `"hh:mm:ss"`.

`__STDC__` The constant 1. It is intended that this identifier be defined to be 1 only in standard-conforming implementations.

`#error` and `#pragma` are new with the ANSI standard; the predefined preprocessor macros are new, but some of them have been available in some implementations.

A13. Grammar

Below is a recapitulation of the grammar that was given throughout the earlier part of this appendix. It has exactly the same content, but is in a different order.

The grammar has undefined terminal symbols *integer-constant*, *character-constant*, *floating-constant*, *identifier*, *string*, and *enumeration-constant*; the `typewriter` style words and symbols are terminals given literally. This grammar can be transformed mechanically into input acceptable to an automatic parser-generator. Besides adding whatever syntactic marking is used to indicate alternatives in productions, it is necessary to expand the "one of" constructions, and (depending on the rules of the parser-generator) to duplicate each production with an *opt* symbol, once with the symbol and once without. With one further change, namely deleting the production *typedef-name: identifier* and making *typedef-name* a terminal symbol, this grammar is acceptable to the YACC parser-generator. It has only one conflict, generated by the `if-else` ambiguity.

translation-unit:
  external-declaration
  translation-unit external-declaration

external-declaration:
  function-definition
  declaration

function-definition:
  declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$ *compound-statement*

declaration:
  declaration-specifiers init-declarator-list$_{opt}$ `;`

declaration-list:
  declaration
  declaration-list declaration

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$

storage-class-specifier: one of
  `auto register static extern typedef`

type-specifier: one of
  `void char short int long float double signed`
  `unsigned` *struct-or-union-specifier enum-specifier typedef-name*

type-qualifier: one of
  `const volatile`

struct-or-union-specifier:
  struct-or-union identifier$_{opt}$ `{` *struct-declaration-list* `}`
  struct-or-union identifier

struct-or-union: one of
  `struct union`

struct-declaration-list:
  struct-declaration
  struct-declaration-list struct-declaration

init-declarator-list:
    init-declarator
    init-declarator-list , *init-declarator*

init-declarator:
    declarator
    declarator = *initializer*

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*

enum-specifier:
    `enum` *identifier*$_{opt}$ { *enumerator-list* }
    `enum` *identifier*

enumerator-list:
    enumerator
    enumerator-list , *enumerator*

enumerator:
    identifier
    identifier = *constant-expression*

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*

initializer:
    assignment-expression
    { *initializer-list* }
    { *initializer-list* , }

initializer-list:
    initializer
    initializer-list , *initializer*

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

statement:
    labeled-statement
    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    jump-statement

labeled-statement:
    identifier : *statement*
    `case` *constant-expression* : *statement*
    `default` : *statement*

expression-statement:
    expression$_{opt}$;

compound-statement:
    { *declaration-list$_{opt}$ statement-list$_{opt}$* }

statement-list:
    statement
    statement-list statement

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*$_{opt}$` ;`

expression:
    assignment-expression
    expression , *assignment-expression*

assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression

assignment-operator: one of
    `=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=`

conditional-expression:
    logical-OR-expression
    logical-OR-expression ? *expression* : *conditional-expression*

constant-expression:
    conditional-expression

logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || *logical-AND-expression*

logical-AND-expression:
    inclusive-OR-expression
    logical-AND-expression && *inclusive-OR-expression*

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

exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression ^ *AND-expression*

AND-expression:
    equality-expression
    AND-expression & *equality-expression*

equality-expression:
    relational-expression
    equality-expression == *relational-expression*
    equality-expression != *relational-expression*

relational-expression:
    shift-expression
    relational-expression < *shift-expression*
    relational-expression > *shift-expression*
    relational-expression <= *shift-expression*
    relational-expression >= *shift-expression*

shift-expression:
  additive-expression
  shift-expression << *additive-expression*
  shift-expression >> *additive-expression*

additive-expression:
  multiplicative-expression
  additive-expression + *multiplicative-expression*
  additive-expression - *multiplicative-expression*

multiplicative-expression:
  cast-expression
  multiplicative-expression * *cast-expression*
  multiplicative-expression / *cast-expression*
  multiplicative-expression % *cast-expression*

cast-expression:
  unary-expression
  (*type-name*) *cast-expression*

unary-expression:
  postfix-expression
  ++ *unary-expression*
  -- *unary-expression*
  unary-operator cast-expression
  `sizeof` *unary-expression*
  `sizeof` (*type-name*)

unary-operator: one of
  & * + - ~ !

postfix-expression:
  primary-expression
  postfix-expression [*expression*]
  postfix-expression (*argument-expression-list*$_{opt}$)
  postfix-expression . *identifier*
  postfix-expression -> *identifier*
  postfix-expression ++
  postfix-expression --

primary-expression:
  identifier
  constant
  string
  (*expression*)

argument-expression-list:
  assignment-expression
  argument-expression-list , *assignment-expression*

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

The following grammar for the preprocessor summarizes the structure of control lines, but is not suitable for mechanized parsing. It includes the symbol *text*, which means ordinary program text, non-conditional preprocessor control lines, or complete preprocessor conditional constructions.

control-line:
  `# define` *identifier token-sequence*
  `# define` *identifier*`(` *identifier* `,` ... `,` *identifier* `)` *token-sequence*
  `# undef` *identifier*
  `# include` <*filename*>
  `# include` "*filename*"
  `# include` *token-sequence*
  `# line` *constant* "*filename*"
  `# line` *constant*
  `# error` *token-sequence*$_{opt}$
  `# pragma` *token-sequence*$_{opt}$
  `#`
  preprocessor-conditional

preprocessor-conditional:
  if-line text elif-parts else-part$_{opt}$ `# endif`

if-line:
  `# if` *constant-expression*
  `# ifdef` *identifier*
  `# ifndef` *identifier*

elif-parts:
  elif-line text
  elif-parts$_{opt}$

elif-line:
  `# elif` *constant-expression*

else-part:
  else-line text

else-line:
  `# else`

APPENDIX B: Standard Library

This appendix is a summary of the library defined by the ANSI standard. The standard library is not part of the C language proper, but an environment that supports standard C will provide the function declarations and type and macro definitions of this library. We have omitted a few functions that are of limited utility or easily synthesized from others; we have omitted multi-byte characters; and we have omitted discussion of locale issues, that is, properties that depend on local language, nationality, or culture.

The functions, types and macros of the standard library are declared in standard *headers*:

<assert.h>	<float.h>	<math.h>	<stdarg.h>	<stdlib.h>
<ctype.h>	<limits.h>	<setjmp.h>	<stddef.h>	<string.h>
<errno.h>	<locale.h>	<signal.h>	<stdio.h>	<time.h>

A header can be accessed by

#include <*header*>

Headers may be included in any order and any number of times. A header must be included outside of any external declaration or definition and before any use of anything it declares. A header need not be a source file.

External identifiers that begin with an underscore are reserved for use by the library, as are all other identifiers that begin with an underscore and an upper-case letter or another underscore.

B1. Input and Output: <stdio.h>

The input and output functions, types, and macros defined in <stdio.h> represent nearly one third of the library.

A *stream* is a source or destination of data that may be associated with a disk or other peripheral. The library supports text streams and binary streams, although on some systems, notably UNIX, these are identical. A text stream is a sequence of lines; each line has zero or more characters and is terminated by '\n'. An environment may need to convert a text stream to or from some other representation (such as mapping '\n' to carriage return and linefeed). A binary stream is a sequence of unprocessed bytes that record internal data, with the property that if it is written, then read back on the same system, it will compare equal.

A stream is connected to a file or device by *opening* it; the connection is broken by *closing* the stream. Opening a file returns a pointer to an object of type `FILE`, which records whatever information is necessary to control the stream. We will use "file pointer" and "stream" interchangeably when there is no ambiguity.

When a program begins execution, the three streams `stdin`, `stdout`, and `stderr` are already open.

B1.1 File Operations

The following functions deal with operations on files. The type `size_t` is the unsigned integral type produced by the `sizeof` operator.

`FILE *fopen(const char *filename, const char *mode)`

`fopen` opens the named file, and returns a stream, or `NULL` if the attempt fails. Legal values for `mode` include

`"r"`	open text file for reading
`"w"`	create text file for writing; discard previous contents if any
`"a"`	append; open or create text file for writing at end of file
`"r+"`	open text file for update (i.e., reading and writing)
`"w+"`	create text file for update; discard previous contents if any
`"a+"`	append; open or create text file for update, writing at end

Update mode permits reading and writing the same file; `fflush` or a file-positioning function must be called between a read and a write or vice versa. If the mode includes `b` after the initial letter, as in `"rb"` or `"w+b"`, that indicates a binary file. Filenames are limited to `FILENAME_MAX` characters. At most `FOPEN_MAX` files may be open at once.

```
FILE *freopen(const char *filename, const char *mode,
              FILE *stream)
```

`freopen` opens the file with the specified mode and associates the stream with it. It returns `stream`, or `NULL` if an error occurs. `freopen` is normally used to change the files associated with `stdin`, `stdout`, or `stderr`.

`int fflush(FILE *stream)`

On an output stream, `fflush` causes any buffered but unwritten data to be written; on an input stream, the effect is undefined. It returns `EOF` for a write error, and zero otherwise. `fflush(NULL)` flushes all output streams.

`int fclose(FILE *stream)`

`fclose` flushes any unwritten data for `stream`, discards any unread buffered input, frees any automatically allocated buffer, then closes the stream. It returns `EOF` if any errors occurred, and zero otherwise.

`int remove(const char *filename)`

`remove` removes the named file, so that a subsequent attempt to open it will fail. It returns non-zero if the attempt fails.

`int rename(const char *oldname, const char *newname)`

`rename` changes the name of a file; it returns non-zero if the attempt fails.

`FILE *tmpfile(void)`
`tmpfile` creates a temporary file of mode `"wb+"` that will be automatically removed when closed or when the program terminates normally. `tmpfile` returns a stream, or `NULL` if it could not create the file.

`char *tmpnam(char s[L_tmpnam])`
`tmpnam(NULL)` creates a string that is not the name of an existing file, and returns a pointer to an internal static array. `tmpnam(s)` stores the string in `s` as well as returning it as the function value; `s` must have room for at least `L_tmpnam` characters. `tmpnam` generates a different name each time it is called; at most `TMP_MAX` different names are guaranteed during execution of the program. Note that `tmpnam` creates a name, not a file.

`int setvbuf(FILE *stream, char *buf, int mode, size_t size)`
`setvbuf` controls buffering for the stream; it must be called before reading, writing, or any other operation. A `mode` of `_IOFBF` causes full buffering, `_IOLBF` line buffering of text files, and `_IONBF` no buffering. If `buf` is not `NULL`, it will be used as the buffer; otherwise a buffer will be allocated. `size` determines the buffer size. `setvbuf` returns non-zero for any error.

`void setbuf(FILE *stream, char *buf)`
If `buf` is `NULL`, buffering is turned off for the stream. Otherwise, `setbuf` is equivalent to `(void) setvbuf(stream, buf, _IOFBF, BUFSIZ)`.

B1.2 Formatted Output

The `printf` functions provide formatted output conversion.

`int fprintf(FILE *stream, const char *format, ...)`
`fprintf` converts and writes output to `stream` under the control of `format`. The return value is the number of characters written, or negative if an error occurred.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to `fprintf`. Each conversion specification begins with the character `%` and ends with a conversion character. Between the `%` and the conversion character there may be, in order:

- Flags (in any order), which modify the specification:

  -, which specifies left adjustment of the converted argument in its field.

  +, which specifies that the number will always be printed with a sign.

  space: if the first character is not a sign, a space will be prefixed.

  0: for numeric conversions, specifies padding to the field width with leading zeros.

  #, which specifies an alternate output form. For o, the first digit will be zero. For x or X, 0x or 0X will be prefixed to a non-zero result. For e, E, f, g, and G, the output will always have a decimal point; for g and G, trailing zeros will not be removed.

- A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally space, but is 0 if the zero padding flag is present.

- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for `e`, `E`, or `f` conversions, or the number of significant digits for `g` or `G` conversion, or the minimum number of digits to be printed for an integer (leading 0s will be added to make up the necessary width).
- A length modifier `h`, `l` (letter ell), or `L`. "`h`" indicates that the corresponding argument is to be printed as a `short` or `unsigned short`; "`l`" indicates that the argument is a `long` or `unsigned long`; "`L`" indicates that the argument is a `long double`.

Width or precision or both may be specified as `*`, in which case the value is computed by converting the next argument(s), which must be `int`.

The conversion characters and their meanings are shown in Table B-1. If the character after the `%` is not a conversion character, the behavior is undefined.

TABLE B-1. PRINTF CONVERSIONS

CHARACTER	ARGUMENT TYPE; CONVERTED TO
`d`, `i`	`int`; signed decimal notation.
`o`	`int`; unsigned octal notation (without a leading zero).
`x`, `X`	`int`; unsigned hexadecimal notation (without a leading `0x` or `0X`), using `abcdef` for `0x` or `ABCDEF` for `0X`.
`u`	`int`; unsigned decimal notation.
`c`	`int`; single character, after conversion to `unsigned char`.
`s`	`char *`; characters from the string are printed until a `'\0'` is reached or until the number of characters indicated by the precision have been printed.
`f`	`double`; decimal notation of the form [−]*mmm.ddd*, where the number of *d*'s is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
`e`, `E`	`double`; decimal notation of the form [−]*m.dddddd*`e`±*xx* or [−]*m.dddddd*`E`±*xx*, where the number of *d*'s is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
`g`, `G`	`double`; `%e` or `%E` is used if the exponent is less than −4 or greater than or equal to the precision; otherwise `%f` is used. Trailing zeros and a trailing decimal point are not printed.
`p`	`void *`; print as a pointer (implementation-dependent representation).
`n`	`int *`; the number of characters written so far by this call to `printf` is *written into* the argument. No argument is converted.
`%`	no argument is converted; print a `%`.

`int printf(const char *format, ...)`

`printf(...)` is equivalent to `fprintf(stdout,...)`.

`int sprintf(char *s, const char *format, ...)`

sprintf is the same as printf except that the output is written into the string s, terminated with '\0'. s must be big enough to hold the result. The return count does not include the '\0'.

`vprintf(const char *format, va_list arg)`
`vfprintf(FILE *stream, const char *format, va_list arg)`
`vsprintf(char *s, const char *format, va_list arg)`

The functions vprintf, vfprintf, and vsprintf are equivalent to the corresponding printf functions, except that the variable argument list is replaced by arg, which has been initialized by the va_start macro and perhaps va_arg calls. See the discussion of <stdarg.h> in Section B7.

B1.3 Formatted Input

The scanf functions deal with formatted input conversion.

`int fscanf(FILE *stream, const char *format, ...)`

fscanf reads from stream under control of format, and assigns converted values through subsequent arguments, *each of which must be a pointer.* It returns when format is exhausted. fscanf returns EOF if end of file or an error occurs before any conversion; otherwise it returns the number of input items converted and assigned.

The format string usually contains conversion specifications, which are used to direct interpretation of input. The format string may contain:

- Blanks or tabs, which are ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of a %, an optional assignment suppression character *, an optional number specifying a maximum field width, an optional h, l, or L indicating the width of the target, and a conversion character.

A conversion specification determines the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by *, as in %*s, however, the input field is simply skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that scanf will read across line boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer. The legal conversion characters are shown in Table B-2.

The conversion characters d, i, n, o, u, and x may be preceded by h if the argument is a pointer to short rather than int, or by l (letter ell) if the argument is a pointer to long. The conversion characters e, f, and g may be preceded by l if a pointer to double rather than float is in the argument list, and by L if a pointer to a long double.

TABLE B-2. SCANF CONVERSIONS

CHARACTER	INPUT DATA; ARGUMENT TYPE
`d`	decimal integer; `int *`.
`i`	integer; `int *`. The integer may be in octal (leading `0`) or hexadecimal (leading `0x` or `0X`).
`o`	octal integer (with or without leading zero); `int *`.
`u`	unsigned decimal integer; `unsigned int *`.
`x`	hexadecimal integer (with or without leading `0x` or `0X`); `int *`.
`c`	characters; `char *`. The next input characters are placed in the indicated array, up to the number given by the width field; the default is 1. No `'\0'` is added. The normal skip over white space characters is suppressed in this case; to read the next non-white space character, use `%1s`.
`s`	string of non-white space characters (not quoted); `char *`, pointing to an array of characters large enough to hold the string and a terminating `'\0'` that will be added.
`e`, `f`, `g`	floating-point number; `float *`. The input format for `float`'s is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an `E` or `e` followed by a possibly signed integer.
`p`	pointer value as printed by `printf("%p")`; `void *`.
`n`	writes into the argument the number of characters read so far by this call; `int *`. No input is read. The converted item count is not incremented.
`[...]`	matches the longest non-empty string of input characters from the set between brackets; `char *`. A `'\0'` is added. `[]...]` includes `]` in the set.
`[^...]`	matches the longest non-empty string of input characters *not* from the set between brackets; `char *`. A `'\0'` is added. `[^]...]` includes `]` in the set.
`%`	literal `%`; no assignment is made.

`int scanf(const char *format, ...)`

`scanf(...)` is identical to `fscanf(stdin,...)`.

`int sscanf(char *s, const char *format, ...)`

`sscanf(s,...)` is equivalent to `scanf(...)` except that the input characters are taken from the string `s`.

B1.4 Character Input and Output Functions

`int fgetc(FILE *stream)`

`fgetc` returns the next character of `stream` as an `unsigned char` (converted to an `int`), or `EOF` if end of file or error occurs.

`char *fgets(char *s, int n, FILE *stream)`
`fgets` reads at most the next `n-1` characters into the array `s`, stopping if a newline is encountered; the newline is included in the array, which is terminated by `'\0'`. `fgets` returns `s`, or `NULL` if end of file or error occurs.

`int fputc(int c, FILE *stream)`
`fputc` writes the character `c` (converted to an `unsigned char`) on `stream`. It returns the character written, or `EOF` for error.

`int fputs(const char *s, FILE *stream)`
`fputs` writes the string `s` (which need not contain `'\n'`) on `stream`; it returns non-negative, or `EOF` for an error.

`int getc(FILE *stream)`
`getc` is equivalent to `fgetc` except that if it is a macro, it may evaluate `stream` more than once.

`int getchar(void)`
`getchar` is equivalent to `getc(stdin)`.

`char *gets(char *s)`
`gets` reads the next input line into the array `s`; it replaces the terminating newline with `'\0'`. It returns `s`, or `NULL` if end of file or error occurs.

`int putc(int c, FILE *stream)`
`putc` is equivalent to `fputc` except that if it is a macro, it may evaluate `stream` more than once.

`int putchar(int c)`
`putchar(c)` is equivalent to `putc(c,stdout)`.

`int puts(const char *s)`
`puts` writes the string `s` and a newline to `stdout`. It returns `EOF` if an error occurs, non-negative otherwise.

`int ungetc(int c, FILE *stream)`
`ungetc` pushes `c` (converted to an `unsigned char`) back onto `stream`, where it will be returned on the next read. Only one character of pushback per stream is guaranteed. `EOF` may not be pushed back. `ungetc` returns the character pushed back, or `EOF` for error.

B1.5 Direct Input and Output Functions

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)`
`fread` reads from `stream` into the array `ptr` at most `nobj` objects of size `size`. `fread` returns the number of objects read; this may be less than the number requested. `feof` and `ferror` must be used to determine status.

```
size_t fwrite(const void *ptr, size_t size, size_t nobj,
                            FILE *stream)
```
`fwrite` writes, from the array `ptr`, `nobj` objects of size `size` on `stream`. It returns the number of objects written, which is less than `nobj` on error.

B1.6 File Positioning Functions

int fseek(FILE *stream, long offset, int origin)
fseek sets the file position for stream; a subsequent read or write will access data beginning at the new position. For a binary file, the position is set to offset characters from origin, which may be SEEK_SET (beginning), SEEK_CUR (current position), or SEEK_END (end of file). For a text stream, offset must be zero, or a value returned by ftell (in which case origin must be SEEK_SET). fseek returns non-zero on error.

long ftell(FILE *stream)
ftell returns the current file position for stream, or -1L on error.

void rewind(FILE *stream)
rewind(fp) is equivalent to fseek(fp,0L,SEEK_SET); clearerr(fp).

int fgetpos(FILE *stream, fpos_t *ptr)
fgetpos records the current position in stream in *ptr, for subsequent use by fsetpos. The type fpos_t is suitable for recording such values. fgetpos returns non-zero on error.

int fsetpos(FILE *stream, const fpos_t *ptr)
fsetpos positions stream at the position recorded by fgetpos in *ptr. fsetpos returns non-zero on error.

B1.7 Error Functions

Many of the functions in the library set status indicators when error or end of file occur. These indicators may be set and tested explicitly. In addition, the integer expression errno (declared in <errno.h>) may contain an error number that gives further information about the most recent error.

void clearerr(FILE *stream)
clearerr clears the end of file and error indicators for stream.

int feof(FILE *stream)
feof returns non-zero if the end of file indicator for stream is set.

int ferror(FILE *stream)
ferror returns non-zero if the error indicator for stream is set.

void perror(const char *s)
perror(s) prints s and an implementation-defined error message corresponding to the integer in errno, as if by

```
fprintf(stderr, "%s: %s\n", s, "error message")
```

See strerror in Section B3.

B2. Character Class Tests: <ctype.h>

The header <ctype.h> declares functions for testing characters. For each function, the argument is an int, whose value must be EOF or representable as an unsigned `char`, and the return value is an `int`. The functions return non-zero (true) if the argument `c` satisfies the condition described, and zero if not.

`isalnum(c)`	`isalpha(c)` or `isdigit(c)` is true
`isalpha(c)`	`isupper(c)` or `islower(c)` is true
`iscntrl(c)`	control character
`isdigit(c)`	decimal digit
`isgraph(c)`	printing character except space
`islower(c)`	lower-case letter
`isprint(c)`	printing character including space
`ispunct(c)`	printing character except space or letter or digit
`isspace(c)`	space, formfeed, newline, carriage return, tab, vertical tab
`isupper(c)`	upper-case letter
`isxdigit(c)`	hexadecimal digit

In the seven-bit ASCII character set, the printing characters are `0x20` (`' '`) to `0x7E` (`'~'`); the control characters are 0 (NUL) to `0x1F` (US), and `0x7F` (DEL).

In addition, there are two functions that convert the case of letters:

`int tolower(int c)`	convert `c` to lower case
`int toupper(int c)`	convert `c` to upper case

If `c` is an upper-case letter, `tolower(c)` returns the corresponding lower-case letter; otherwise it returns `c`. If `c` is a lower-case letter, `toupper(c)` returns the corresponding upper-case letter; otherwise it returns `c`.

B3. String Functions: <string.h>

There are two groups of string functions defined in the header `<string.h>`. The first have names beginning with `str`; the second have names beginning with `mem`. Except for `memmove`, the behavior is undefined if copying takes place between overlapping objects. Comparison functions treat arguments as `unsigned char` arrays.

In the following table, variables `s` and `t` are of type `char *`; `cs` and `ct` are of type `const char *`; `n` is of type `size_t`; and `c` is an `int` converted to `char`.

`char *strcpy(s,ct)`	copy string `ct` to string `s`, including `'\0'`; return `s`.
`char *strncpy(s,ct,n)`	copy at most `n` characters of string `ct` to `s`; return `s`. Pad with `'\0'`s if `t` has fewer than `n` characters.
`char *strcat(s,ct)`	concatenate string `ct` to end of string `s`; return `s`.
`char *strncat(s,ct,n)`	concatenate at most `n` characters of string `ct` to string `s`, terminate `s` with `'\0'`; return `s`.
`int strcmp(cs,ct)`	compare string `cs` to string `ct`; return <0 if `cs<ct`, 0 if `cs==ct`, or >0 if `cs>ct`.
`int strncmp(cs,ct,n)`	compare at most `n` characters of string `cs` to string `ct`; return <0 if `cs<ct`, 0 if `cs==ct`, or >0 if `cs>ct`.
`char *strchr(cs,c)`	return pointer to first occurrence of `c` in `cs` or `NULL` if not present.
`char *strrchr(cs,c)`	return pointer to last occurrence of `c` in `cs` or `NULL` if not present.

`size_t strspn(cs,ct)`	return length of prefix of `cs` consisting of characters in `ct`.
`size_t strcspn(cs,ct)`	return length of prefix of `cs` consisting of characters *not* in `ct`.
`char *strpbrk(cs,ct)`	return pointer to first occurrence in string `cs` of any character of string `ct`, or `NULL` if none are present.
`char *strstr(cs,ct)`	return pointer to first occurrence of string `ct` in `cs`, or `NULL` if not present.
`size_t strlen(cs)`	return length of `cs`.
`char *strerror(n)`	return pointer to implementation-defined string corresponding to error `n`.
`char *strtok(s,ct)`	`strtok` searches `s` for tokens delimited by characters from `ct`; see below.

A sequence of calls of `strtok(s,ct)` splits `s` into tokens, each delimited by a character from `ct`. The first call in a sequence has a non-`NULL` `s`. It finds the first token in `s` consisting of characters not in `ct`; it terminates that by overwriting the next character of `s` with `'\0'` and returns a pointer to the token. Each subsequent call, indicated by a `NULL` value of `s`, returns the next such token, searching from just past the end of the previous one. `strtok` returns `NULL` when no further token is found. The string `ct` may be different on each call.

The `mem`... functions are meant for manipulating objects as character arrays; the intent is an interface to efficient routines. In the following table, `s` and `t` are of type `void *`; `cs` and `ct` are of type `const void *`; `n` is of type `size_t`; and `c` is an `int` converted to an `unsigned char`.

`void *memcpy(s,ct,n)`	copy `n` characters from `ct` to `s`, and return `s`.
`void *memmove(s,ct,n)`	same as `memcpy` except that it works even if the objects overlap.
`int memcmp(cs,ct,n)`	compare the first `n` characters of `cs` with `ct`; return as with `strcmp`.
`void *memchr(cs,c,n)`	return pointer to first occurrence of character `c` in `cs`, or `NULL` if not present among the first `n` characters.
`void *memset(s,c,n)`	place character `c` into first `n` characters of `s`, return `s`.

B4. Mathematical Functions: <math.h>

The header `<math.h>` declares mathematical functions and macros.

The macros `EDOM` and `ERANGE` (found in `<errno.h>`) are non-zero integral constants that are used to signal domain and range errors for the functions; `HUGE_VAL` is a positive `double` value. A *domain error* occurs if an argument is outside the domain over which the function is defined. On a domain error, `errno` is set to `EDOM`; the return value is implementation-dependent. A *range error* occurs if the result of the function cannot be represented as a `double`. If the result overflows, the function returns `HUGE_VAL` with the right sign, and `errno` is set to `ERANGE`. If the result underflows, the function returns zero; whether `errno` is set to `ERANGE` is implementation-defined.

In the following table, `x` and `y` are of type `double`, `n` is an `int`, and all functions return `double`. Angles for trigonometric functions are expressed in radians.

`sin(x)`	sine of x
`cos(x)`	cosine of x
`tan(x)`	tangent of x
`asin(x)`	$\sin^{-1}(x)$ in range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.
`acos(x)`	$\cos^{-1}(x)$ in range $[0, \pi]$, $x \in [-1, 1]$.
`atan(x)`	$\tan^{-1}(x)$ in range $[-\pi/2, \pi/2]$.
`atan2(y,x)`	$\tan^{-1}(y/x)$ in range $[-\pi, \pi]$.
`sinh(x)`	hyperbolic sine of x
`cosh(x)`	hyperbolic cosine of x
`tanh(x)`	hyperbolic tangent of x
`exp(x)`	exponential function e^x
`log(x)`	natural logarithm $\ln(x)$, $x>0$.
`log10(x)`	base 10 logarithm $\log_{10}(x)$, $x>0$.
`pow(x,y)`	x^y. A domain error occurs if `x`=0 and `y`$\leqslant$0, or if `x`<0 and `y` is not an integer.
`sqrt(x)`	$\sqrt{x}$, $x \geqslant 0$.
`ceil(x)`	smallest integer not less than `x`, as a `double`.
`floor(x)`	largest integer not greater than `x`, as a `double`.
`fabs(x)`	absolute value $\lvert x \rvert$
`ldexp(x,n)`	$x \cdot 2^n$
`frexp(x, int *exp)`	splits `x` into a normalized fraction in the interval [1/2, 1), which is returned, and a power of 2, which is stored in `*exp`. If `x` is zero, both parts of the result are zero.
`modf(x, double *ip)`	splits `x` into integral and fractional parts, each with the same sign as `x`. It stores the integral part in `*ip`, and returns the fractional part.
`fmod(x,y)`	floating-point remainder of `x/y`, with the same sign as `x`. If `y` is zero, the result is implementation-defined.

B5. Utility Functions: <stdlib.h>

The header `<stdlib.h>` declares functions for number conversion, storage allocation, and similar tasks.

`double atof(const char *s)`

`atof` converts `s` to `double`; it is equivalent to `strtod(s, (char**)NULL)`.

`int atoi(const char *s)`

converts `s` to `int`; it is equivalent to `(int)strtol(s, (char**)NULL, 10)`.

`long atol(const char *s)`

converts `s` to `long`; it is equivalent to `strtol(s, (char**)NULL, 10)`.

`double strtod(const char *s, char **endp)`

`strtod` converts the prefix of `s` to `double`, ignoring leading white space; it stores a pointer to any unconverted suffix in `*endp` unless `endp` is `NULL`. If the answer would overflow, `HUGE_VAL` is returned with the proper sign; if the answer would underflow, zero is returned. In either case `errno` is set to `ERANGE`.

`long strtol(const char *s, char **endp, int base)`
`strtol` converts the prefix of `s` to `long`, ignoring leading white space; it stores a pointer to any unconverted suffix in `*endp` unless `endp` is `NULL`. If `base` is between 2 and 36, conversion is done assuming that the input is written in that base. If `base` is zero, the base is 8, 10, or 16; leading 0 implies octal and leading `0x` or `0X` hexadecimal. Letters in either case represent digits from 10 to `base-1`; a leading `0x` or `0X` is permitted in base 16. If the answer would overflow, `LONG_MAX` or `LONG_MIN` is returned, depending on the sign of the result, and `errno` is set to `ERANGE`.

`unsigned long strtoul(const char *s, char **endp, int base)`
`strtoul` is the same as `strtol` except that the result is `unsigned long` and the error value is `ULONG_MAX`.

`int rand(void)`
`rand` returns a pseudo-random integer in the range 0 to `RAND_MAX`, which is at least 32767.

`void srand(unsigned int seed)`
`srand` uses `seed` as the seed for a new sequence of pseudo-random numbers. The initial seed is 1.

`void *calloc(size_t nobj, size_t size)`
`calloc` returns a pointer to space for an array of `nobj` objects, each of size `size`, or `NULL` if the request cannot be satisfied. The space is initialized to zero bytes.

`void *malloc(size_t size)`
`malloc` returns a pointer to space for an object of size `size`, or `NULL` if the request cannot be satisfied. The space is uninitialized.

`void *realloc(void *p, size_t size)`
`realloc` changes the size of the object pointed to by `p` to `size`. The contents will be unchanged up to the minimum of the old and new sizes. If the new size is larger, the new space is uninitialized. `realloc` returns a pointer to the new space, or `NULL` if the request cannot be satisfied, in which case `*p` is unchanged.

`void free(void *p)`
`free` deallocates the space pointed to by `p`; it does nothing if `p` is `NULL`. `p` must be a pointer to space previously allocated by `calloc`, `malloc`, or `realloc`.

`void abort(void)`
`abort` causes the program to terminate abnormally, as if by `raise(SIGABRT)`.

`void exit(int status)`
`exit` causes normal program termination. `atexit` functions are called in reverse order of registration, open files are flushed, open streams are closed, and control is returned to the environment. How `status` is returned to the environment is implementation-dependent, but zero is taken as successful termination. The values `EXIT_SUCCESS` and `EXIT_FAILURE` may also be used.

`int atexit(void (*fcn)(void))`

`atexit` registers the function `fcn` to be called when the program terminates normally; it returns non-zero if the registration cannot be made.

`int system(const char *s)`

`system` passes the string `s` to the environment for execution. If `s` is `NULL`, `system` returns non-zero if there is a command processor. If `s` is not `NULL`, the return value is implementation-dependent.

`char *getenv(const char *name)`

`getenv` returns the environment string associated with `name`, or `NULL` if no string exists. Details are implementation-dependent.

```
void *bsearch(const void *key, const void *base,
      size_t n, size_t size,
      int (*cmp)(const void *keyval, const void *datum))
```

`bsearch` searches `base[0]`...`base[n-1]` for an item that matches `*key`. The function `cmp` must return negative if its first argument (the search key) is less than its second (a table entry), zero if equal, and positive if greater. Items in the array `base` must be in ascending order. `bsearch` returns a pointer to a matching item, or `NULL` if none exists.

```
void qsort(void *base, size_t n, size_t size,
                   int (*cmp)(const void *, const void *))
```

`qsort` sorts into ascending order an array `base[0]`...`base[n-1]` of objects of size `size`. The comparison function `cmp` is as in `bsearch`.

`int abs(int n)`

`abs` returns the absolute value of its `int` argument.

`long labs(long n)`

`labs` returns the absolute value of its `long` argument.

`div_t div(int num, int denom)`

`div` computes the quotient and remainder of `num/denom`. The results are stored in the `int` members `quot` and `rem` of a structure of type `div_t`.

`ldiv_t ldiv(long num, long denom)`

`div` computes the quotient and remainder of `num/denom`. The results are stored in the `long` members `quot` and `rem` of a structure of type `ldiv_t`.

B6. Diagnostics: <assert.h>

The `assert` macro is used to add diagnostics to programs:

`void assert(int` *expression*`)`

If *expression* is zero when

`assert(`*expression*`)`

is executed, the `assert` macro will print on `stderr` a message, such as

`Assertion failed:` *expression*`, file` *filename*`, line` *nnn*

It then calls `abort` to terminate execution. The source filename and line number come from the preprocessor macros __FILE__ and __LINE__.

If NDEBUG is defined at the time <assert.h> is included, the assert macro is ignored.

B7. Variable Argument Lists: <stdarg.h>

The header <stdarg.h> provides facilities for stepping through a list of function arguments of unknown number and type.

Suppose *lastarg* is the last named parameter of a function f with a variable number of arguments. Then declare within f a variable ap of type va_list that will point to each argument in turn:

```
va_list ap;
```

ap must be initialized once with the macro va_start before any unnamed argument is accessed:

```
va_start(va_list ap, lastarg);
```

Thereafter, each execution of the macro va_arg will produce a value that has the type and value of the next unnamed argument, and will also modify ap so the next use of va_arg returns the next argument:

```
type va_arg(va_list ap, type);
```

The macro

```
void va_end(va_list ap);
```

must be called once after the arguments have been processed but before f is exited.

B8. Non-local Jumps: <setjmp.h>

The declarations in <setjmp.h> provide a way to avoid the normal function call and return sequence, typically to permit an immediate return from a deeply nested function call.

int setjmp(jmp_buf env)

The macro setjmp saves state information in env for use by longjmp. The return is zero from a direct call of setjmp, and non-zero from a subsequent call of longjmp. A call to setjmp can only occur in certain contexts, basically the test of if, switch, and loops, and only in simple relational expressions.

```
if (setjmp(env) == 0)
    /* get here on direct call */
else
    /* get here by calling longjmp */
```

void longjmp(jmp_buf env, int val)

longjmp restores the state saved by the most recent call to setjmp, using information saved in env, and execution resumes as if the setjmp function had just executed and returned the non-zero value val. The function containing the setjmp must not have terminated. Accessible objects have the values they had when longjmp was called, except that non-volatile automatic variables in the function calling setjmp become undefined if they were changed after the setjmp call.

B9. Signals: <signal.h>

The header `<signal.h>` provides facilities for handling exceptional conditions that arise during execution, such as an interrupt signal from an external source or an error in execution.

`void (*signal(int sig, void (*handler)(int)))(int)`

`signal` determines how subsequent signals will be handled. If `handler` is `SIG_DFL`, the implementation-defined default behavior is used; if it is `SIG_IGN`, the signal is ignored; otherwise, the function pointed to by `handler` will be called, with the argument of the type of signal. Valid signals include

`SIGABRT`	abnormal termination, e.g., from `abort`
`SIGFPE`	arithmetic error, e.g., zero divide or overflow
`SIGILL`	illegal function image, e.g., illegal instruction
`SIGINT`	interactive attention, e.g., interrupt
`SIGSEGV`	illegal storage access, e.g., access outside memory limits
`SIGTERM`	termination request sent to this program

`signal` returns the previous value of `handler` for the specific signal, or `SIG_ERR` if an error occurs.

When a signal `sig` subsequently occurs, the signal is restored to its default behavior; then the signal-handler function is called, as if by `(*handler)(sig)`. If the handler returns, execution will resume where it was when the signal occurred.

The initial state of signals is implementation-defined.

`int raise(int sig)`

`raise` sends the signal `sig` to the program; it returns non-zero if unsuccessful.

B10. Date and Time Functions: <time.h>

The header `<time.h>` declares types and functions for manipulating date and time. Some functions process *local time*, which may differ from calendar time, for example because of time zone. `clock_t` and `time_t` are arithmetic types representing times, and `struct tm` holds the components of a calendar time:

`int tm_sec;`	seconds after the minute (0, 61)
`int tm_min;`	minutes after the hour (0, 59)
`int tm_hour;`	hours since midnight (0, 23)
`int tm_mday;`	day of the month (1, 31)
`int tm_mon;`	months *since* January (0, 11)
`int tm_year;`	years since 1900
`int tm_wday;`	days since Sunday (0, 6)
`int tm_yday;`	days since January 1 (0, 365)
`int tm_isdst;`	Daylight Saving Time flag

`tm_isdst` is positive if Daylight Saving Time is in effect, zero if not, and negative if the information is not available.

`clock_t clock(void)`

`clock` returns the processor time used by the program since the beginning of execution, or -1 if unavailable. `clock()/CLOCKS_PER_SEC` is a time in seconds.

`time_t time(time_t *tp)`
`time` returns the current calendar time or `-1` if the time is not available. If `tp` is not `NULL`, the return value is also assigned to `*tp`.

`double difftime(time_t time2, time_t time1)`
`difftime` returns `time2-time1` expressed in seconds.

`time_t mktime(struct tm *tp)`
`mktime` converts the local time in the structure `*tp` into calendar time in the same representation used by `time`. The components will have values in the ranges shown. `mktime` returns the calendar time or `-1` if it cannot be represented.

The next four functions return pointers to static objects that may be overwritten by other calls.

`char *asctime(const struct tm *tp)`
`asctime` converts the time in the structure `*tp` into a string of the form

```
Sun Jan  3 15:14:13 1988\n\0
```

`char *ctime(const time_t *tp)`
`ctime` converts the calendar time `*tp` to local time; it is equivalent to

```
asctime(localtime(tp))
```

`struct tm *gmtime(const time_t *tp)`
`gmtime` converts the calendar time `*tp` into Coordinated Universal Time (UTC). It returns `NULL` if UTC is not available. The name `gmtime` has historical significance.

`struct tm *localtime(const time_t *tp)`
`localtime` converts the calendar time `*tp` into local time.

```
size_t strftime(char *s, size_t smax, const char *fmt,
                const struct tm *tp)
```
`strftime` formats date and time information from `*tp` into `s` according to `fmt`, which is analogous to a `printf` format. Ordinary characters (including the terminating `'\0'`) are copied into `s`. Each `%c` is replaced as described below, using values appropriate for the local environment. No more than `smax` characters are placed into `s`. `strftime` returns the number of characters, excluding the `'\0'`, or zero if more than `smax` characters were produced.

`%a`	abbreviated weekday name.
`%A`	full weekday name.
`%b`	abbreviated month name.
`%B`	full month name.
`%c`	local date and time representation.
`%d`	day of the month (01-31).
`%H`	hour (24-hour clock) (00-23).
`%I`	hour (12-hour clock) (01-12).
`%j`	day of the year (001-366).

`%m`	month (01-12).
`%M`	minute (00-59).
`%p`	local equivalent of AM or PM.
`%S`	second (00-61).
`%U`	week number of the year (Sunday as 1st day of week) (00-53).
`%w`	weekday (0-6, Sunday is 0).
`%W`	week number of the year (Monday as 1st day of week) (00-53).
`%x`	local date representation.
`%X`	local time representation.
`%y`	year without century (00-99).
`%Y`	year with century.
`%Z`	time zone name, if any.
`%%`	%.

B11. Implementation-defined Limits: <limits.h> and <float.h>

The header `<limits.h>` defines constants for the sizes of integral types. The values below are acceptable minimum magnitudes; larger values may be used.

CHAR_BIT	8	bits in a `char`
CHAR_MAX	UCHAR_MAX *or* SCHAR_MAX	maximum value of `char`
CHAR_MIN	0 *or* SCHAR_MIN	minimum value of `char`
INT_MAX	+32767	maximum value of `int`
INT_MIN	-32767	minimum value of `int`
LONG_MAX	+2147483647	maximum value of `long`
LONG_MIN	-2147483647	minimum value of `long`
SCHAR_MAX	+127	maximum value of `signed char`
SCHAR_MIN	-127	minimum value of `signed char`
SHRT_MAX	+32767	maximum value of `short`
SHRT_MIN	-32767	minimum value of `short`
UCHAR_MAX	255	maximum value of `unsigned char`
UINT_MAX	65535	maximum value of `unsigned int`
ULONG_MAX	4294967295	maximum value of `unsigned long`
USHRT_MAX	65535	maximum value of `unsigned short`

The names in the table below, a subset of `<float.h>`, are constants related to floating-point arithmetic. When a value is given, it represents the minimum magnitude for the corresponding quantity. Each implementation defines appropriate values.

FLT_RADIX	2	radix of exponent representation, e.g., 2, 16
FLT_ROUNDS		floating-point rounding mode for addition
FLT_DIG	6	decimal digits of precision
FLT_EPSILON	1E-5	smallest number x such that $1.0 + x \neq 1.0$
FLT_MANT_DIG		number of base FLT_RADIX digits in mantissa
FLT_MAX	1E+37	maximum floating-point number
FLT_MAX_EXP		maximum n such that $\mathtt{FLT\_RADIX}^n - 1$ is representable
FLT_MIN	1E-37	minimum normalized floating-point number
FLT_MIN_EXP		minimum n such that 10^n is a normalized number

`DBL_DIG`	10	decimal digits of precision
`DBL_EPSILON`	1E-9	smallest number x such that $1.0 + x \neq 1.0$
`DBL_MANT_DIG`		number of base `FLT_RADIX` digits in mantissa
`DBL_MAX`	1E+37	maximum `double` floating-point number
`DBL_MAX_EXP`		maximum n such that `FLT_RADIX`$^n-1$ is representable
`DBL_MIN`	1E-37	minimum normalized `double` floating-point number
`DBL_MIN_EXP`		minimum n such that 10^n is a normalized number

APPENDIX C: Summary of Changes

Since the publication of the first edition of this book, the definition of the C language has undergone changes. Almost all were extensions of the original language, and were carefully designed to remain compatible with existing practice; some repaired ambiguities in the original description; and some represent modifications that change existing practice. Many of the new facilities were announced in the documents accompanying compilers available from AT&T, and have subsequently been adopted by other suppliers of C compilers. More recently, the ANSI committee standardizing the language incorporated most of these changes, and also introduced other significant modifications. Their report was in part anticipated by some commercial compilers even before issuance of the formal C standard.

This Appendix summarizes the differences between the language defined by the first edition of this book, and that expected to be defined by the final Standard. It treats only the language itself, not its environment and library; although these are an important part of the Standard, there is little to compare with, because the first edition did not attempt to prescribe an environment or library.

- Preprocessing is more carefully defined in the Standard than in the first edition, and is extended: it is explicitly token based; there are new operators for catenation of tokens (`##`), and creation of strings (`#`); there are new control lines like `#elif` and `#pragma`; redeclaration of macros by the same token sequence is explicitly permitted; parameters inside strings are no longer replaced. Splicing of lines by \ is permitted everywhere, not just in strings and macro definitions. See §A12.
- The minimum significance of all internal identifiers is increased to 31 characters; the smallest mandated significance of identifiers with external linkage remains 6 monocase letters. (Many implementations provide more.)
- Trigraph sequences introduced by `??` allow representation of characters lacking in some character sets. Escapes for `#\^[]{}|~` are defined; see §A12.1. Observe that the introduction of trigraphs may change the meaning of strings containing the sequence `??`.
- New keywords (`void`, `const`, `volatile`, `signed`, `enum`) are introduced. The stillborn `entry` keyword is withdrawn.
- New escape sequences, for use within character constants and string literals, are defined. The effect of following \ by a character not part of an approved escape sequence is undefined. See §A2.5.2.

- Everyone's favorite trivial change: `8` and `9` are not octal digits.
- The Standard introduces a larger set of suffixes to make the type of constants explicit: `U` or `L` for integers, `F` or `L` for floating. It also refines the rules for the type of unsuffixed constants (§A2.5).
- Adjacent string literals are concatenated.
- There is a notation for wide-character string literals and character constants; see §A2.6.
- Characters, as well as other types, may be explicitly declared to carry, or not to carry, a sign by using the keywords `signed` or `unsigned`. The locution `long float` as a synonym for `double` is withdrawn, but `long double` may be used to declare an extra-precision floating quantity.
- For some time, type `unsigned char` has been available. The standard introduces the `signed` keyword to make signedness explicit for `char` and other integral objects.
- The `void` type has been available in most implementations for some years. The Standard introduces the use of the `void *` type as a generic pointer type; previously `char *` played this role. At the same time, explicit rules are enacted against mixing pointers and integers, and pointers of different type, without the use of casts.
- The Standard places explicit minima on the ranges of the arithmetic types, and mandates headers (`<limits.h>` and `<float.h>`) giving the characteristics of each particular implementation.
- Enumerations are new since the first edition of this book.
- The Standard adopts from C++ the notion of type qualifier, for example `const` (§A8.2).
- Strings are no longer modifiable, and so may be placed in read-only memory.
- The "usual arithmetic conversions" are changed, essentially from "for integers, `unsigned` always wins; for floating point, always use `double`" to "promote to the smallest capacious-enough type." See §A6.5.
- The old assignment operators like `=+` are truly gone. Also, assignment operators are now single tokens; in the first edition, they were pairs, and could be separated by white space.
- A compiler's license to treat mathematically associative operators as computationally associative is revoked.
- A unary `+` operator is introduced for symmetry with unary `-`.
- A pointer to a function may be used as a function designator without an explicit `*` operator. See §A7.3.2.
- Structures may be assigned, passed to functions, and returned by functions.
- Applying the address-of operator to arrays is permitted, and the result is a pointer to the array.
- The `sizeof` operator, in the first edition, yielded type `int`; subsequently, many implementations made it `unsigned`. The Standard makes its type explicitly implementation-dependent, but requires the type, `size_t`, to be defined in a

standard header (`<stddef.h>`). A similar change occurs in the type (`ptrdiff_t`) of the difference between pointers. See §A7.4.8 and §A7.7.

- The address-of operator `&` may not be applied to an object declared `register`, even if the implementation chooses not to keep the object in a register.
- The type of a shift expression is that of the left operand; the right operand can't promote the result. See §A7.8.
- The Standard legalizes the creation of a pointer just beyond the end of an array, and allows arithmetic and relations on it; see §A7.7.
- The Standard introduces (borrowing from C++) the notion of a function prototype declaration that incorporates the types of the parameters, and includes an explicit recognition of variadic functions together with an approved way of dealing with them. See §§A7.3.2, A8.6.3, B7. The older style is still accepted, with restrictions.
- Empty declarations, which have no declarators and don't declare at least a structure, union, or enumeration, are forbidden by the Standard. On the other hand, a declaration with just a structure or union tag redeclares that tag even if it was declared in an outer scope.
- External data declarations without any specifiers or qualifiers (just a naked declarator) are forbidden.
- Some implementations, when presented with an `extern` declaration in an inner block, would export the declaration to the rest of the file. The Standard makes it clear that the scope of such a declaration is just the block.
- The scope of parameters is injected into a function's compound statement, so that variable declarations at the top level of the function cannot hide the parameters.
- The name spaces of identifiers are somewhat different. The Standard puts all tags in a single name space, and also introduces a separate name space for labels; see §A11.1. Also, member names are associated with the structure or union of which they are a part. (This has been common practice from some time.)
- Unions may be initialized; the initializer refers to the first member.
- Automatic structures, unions, and arrays may be initialized, albeit in a restricted way.
- Character arrays with an explicit size may be initialized by a string literal with exactly that many characters (the `\0` is quietly squeezed out).
- The controlling expression, and the case labels, of a switch may have any integral type.

Index