ANSI X3.159-1989

American National Standard

for Information Systems –

Programming Language – C

ANSI X3.159-1989

ANSI **American National Standards Institute**
1430 Broadway
New York , New York
10018

This standard has been adopted for Federal Government use.

Details concerning its use within the Federal Government are contained in Federal Information Processing Standards Publication 160, C. For a complete list of the publications available in the Federal Information Processing Standards Series, write to the Standards Processing Coordinator (ADP), National Institute of Standards and Technology, Gaithersburg, MD 20899.

American National Standard
for Information Systems –

Programming Language –
C

Secretariat

Computer and Business Equipment Manufacturers Association

Approved December 14, 1989

American National Standards Institute, Inc

Abstract

This standard specifies the form and establishes the interpretation of programs expressed in the programming language C. Its purpose is to promote portability, reliability, maintainability, and efficient execution of C language programs on a variety of computing systems.

Sections are included that detail the C language itself and the contents of the C-language execution library. Appendixes summarize aspects of both of them, and enumerate factors that influence the portability of C programs.

Although this standard is intended to guide knowledgeable C-language programmers as well as implementators of C-language translation systems, the document itself is not designed to serve as a tutorial.

American National Standard

Approval of an American National Standard requires verification by ANSI that the requirements for due process, consensus, and other criteria for approval have been met by the standards developer.

Consensus is established when, in the judgment of the ANSI Board of Standards Review, substantial agreement has been reached by directly and materially affected interests. Substantial agreement means much more than a simple majority, but not necessarily unanimity. Consensus requires that all views and objections be considered, and that a concerted effort be made toward their resolution.

The use of American National Standards is completely voluntary; their existence does not in any respect preclude anyone, whether he has approved the standards or not, from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standards.

The American National Standards Institute does not develop standards and will in no circumstances give an interpretation of any American National Standard. Moreover, no person shall have the right or authority to issue an interpretation of an American National Standard in the name of the American National Standards Institute. Requests for interpretations should be addressed to the secretariat or sponsor whose name appears on the title page of this standard.

CAUTION NOTICE: This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken periodically to reaffirm, revise, or withdraw this standard. Purchasers of American National Standards may receive current information on all standards by calling or writing the American National Standards Institute.

Foreword

This standard specifies the syntax and semantics of programs written in the C programming language. It specifies the C program's interactions with the execution environment via input and output data. It also specifies restrictions and limits imposed upon conforming implementations of C language translators.

The standard was developed by the X3J11 Technical Committee on the C Programming Language under project 381-D by American National Standards Committee on Computers and Information Processing (X3). SPARC document number 83-079 describes the purpose of this project to "provide an unambiguous and machine-independent definition of the language C."

The need for a single clearly defined standard had arisen in the C community due to a rapidly expanding use of the C programming language and the variety of differing translator implementations that had been and were being developed. The existence of similar but incompatible implementations was a serious problem for program developers who wished to develop code that would compile and execute as expected in several different environments.

Part of this problem could be traced to the fact that implementors did not have an adequate definition of the C language upon which to base their implementations. The de facto C programming language standard, *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, is an excellent book; however, it is not precise or complete enough to specify the C language fully. In addition, the language has grown over years of use to incorporate new ideas in programming and to address some of the weaknesses of the original language.

American National Standard Programming Language C addresses the problems of both the program developer and the translator implementor by specifying the C language precisely.

The work of X3J11 began in the summer of 1983, based on the several documents that were made available to the Committee (see 1.5, Base Documents). The Committee divided the effort into three pieces: the environment, the language, and the library. A complete specification in each of these areas is necessary if truly portable programs are to be developed. Each of these areas is addressed in the standard. The Committee evaluated many proposals for additions, deletions, and changes to the base documents during its deliberations. A concerted effort was made to codify existing practice wherever unambiguous and consistent practice could be identified. However, where no consistent practice could be identified, the Committee worked to establish clear rules that were consistent with the overall flavor of the language.

This document was approved as an American National Standard by the American National Standards Institute (ANSI) on December 14, 1989.

Suggestions for improvement of this standard are welcome. They should be sent to the Computer and Business Equipment Manufacturers Association, 311 First Street, N.W., Suite 500, Washington, DC 20001-2178.

The standard was processed and approved for submittal to ANSI by the Accredited Standards Committee on Information Processing Systems, X3. Committee approval of the standard does not necessarily imply that all members voted for its approval. At the time that it approved this standard, the X3 Committee had the following members:

Richard Gibson, Chair
Donald C. Loughry, Vice-Chair
(Vacant), Administrative Secretary

Organization Represented	*Name of Representative*
Allen-Bradley	Ronald H. Reimer
American Library Association	Paul E. Peters
American Nuclear Society	Geraldine C. Main
AMP, Inc.	Edward R. Kelly
	Ronald Lloyd (Alt)
Apple Computer, Inc.	Karen Higginbottom
	Michael J. Lawler (Alt)
Association of the Institute for Certification of Computer Professionals	Thomas M. Kurihara
AT&T	Thomas F. Frost
	Paul D. Bartoli (Alt)
Boeing Company	Paul W. Mercer
Compaq Computer Corporation	James L. Barnes
Control Data Corporation	Ernest L. Fogle
Cooperating Users of Burroughs Equipment	Thomas Easterday
	Donald Miller (Alt)
Dataproducts Corporation	Charles D. Card
Digital Equipment Computer Users Society	James R. Ebright
Digital Equipment Corporation	Gary S. Robinson
	Delbert L. Shoemaker (Alt)
Eastman Kodak	Gary Haines
	James D. Converse (Alt)
Electronic Data Systems Corporation	Jerrold S. Foley
GUIDE International	Frank Kirshenbaum
	Jeffery Roberts (Alt)
Hewlett-Packard	Donald C. Loughry
Honeywell Bull	David M. Taylor
IBM Corporation	Robert H. Follett
	Mary Anne Gray (Alt)
IEEE Computer Society	Tom Hannon
	Bob Pritchard (Alt)
Lawrence Berkeley Laboratory	David F. Stevens
	Robert L. Fink (Alt)
MAP/TOP	Michael Kaminski
Moore Business Forms	Delmer H. Oddy
National Communications System	Dennis Bodson
	Donald Wilson (Alt)
National Institute of Standards and Technology	Robert E. Rountree
	Michael D. Hogan (Alt)
NCR Corporation	Thomas W. Kern
	A. R. Daniels (Alt)
OMNICOM	Harold C. Folts
	Cheryl C. Slobodian (Alt)
Prime Computer, Inc.	Thomas Connerty
	Phillip Cieply (Alt)
Recognition Technology Users Association	Herbert F. Schantz
SHARE, Inc.	Thomas B. Steel Jr.
	Gary Ainsworth (Alt)
3M Company	Paul D. Jaimke
Unisys	Marvin W. Bass
	Steven P. Oksala (Alt)
U.S. Department of Defense	William C. Rinehuls
	Thomas M. Kurihara (Alt)
U.S. General Services Administration	Dale O. Christensen
	Larry L. Jackson (Alt)
US WEST	Gary Dempsey
	Susan Capraro (Alt)
VIM	Chris Tanner
	John Ulrich (Alt)

Organization Represented	*Name of Representative*
Wang Corporation	J. J. Cinecoe
	Sarah Wagner (Alt)
Wintergreen Information Services	John L. Wheeler
Xerox Corporation	Roy Pierce

Technical Committee X3J11 on the C Programming Language had the following members at the time they forwarded this document to X3 for processing as an American National Standard:

Jim Brodie, Chair

Thomas Plum, Vice-Chair

P. J. Plauger, Secretary

P. J. Plauger, International Representative (previously: Steve Hersee)

Andrew Johnson, Vocabulary Representative

David F. Prosser, Draft Redactor (previously: Lawrence Rosler)

Randy Hudson, Rationale Redactor

Ralph Ryan; Ralph Phraner, Environment Subcommittee Chairs

Lawrence Rosler, Language Subcommittee Chair

P. J. Plauger, Library Subcommittee Chair

Organization Represented	*Name of Representative*
AT&T	David F. Prosser
	Steven J. Adamski, X3H2 SQL liaison (Alt)
Alliant Computer Systems	Kevin Brosnan
Amdahl	Neal Weidenhofer
American Cimflex	Philip C. Steel
	Eric McGlohon (Alt)
Amoco Production Company	Tracy Pipkin
	William Allen (Alt)
Analog Devices	Stephen Kafka
	Kevin Leary (Alt)
	Gordon Sterling (Alt)
Apollo Computer	John Peyton
Apple Computer, Inc.	Elizabeth Crockett
Arinc	Ed Wells
	Tom Ketterhagen (Alt)
Aspen Scientific	Vaughn Vernon
Bell Communications Research	Craig Bordelon
	Steve Carter (Alt)
	William Puig (Alt)
Borland International	Bob Jervis
Boston Systems Office	Yom-Tov Meged
	Rose Thomson (Alt)
COSMIC	Maurice Fathi
Charles River Data Systems	John Wu
Chemical Abstracts Service	Daniel Mickey
	Thomas Mimlitch (Alt)
Chicago Research & Trading Group	Alan Losoff
Citibank	Edward Briggs
Cobra S/A	Firmo Freire
Cognos	Jim Patterson
Columbia U. Center for Computing	Bruce Tetelman
CompuDas	Terry Moore
Computer Associates	Mark Barrenechea
Computer Innovations	George Eberhardt
	Dave Neathery (Alt)
Computrition	Joseph Bibbo
Concurrent Computer Corporation	Steve Davies
Control Data	Don Fosbury
	George VandeBunte (Alt)
Cormorant Communications	Lloyd Irons

Organization Represented	*Name of Representative*
Cray Research	Tom MacDonald
	Lynne Johnson (Alt)
	Dave Becker (Alt)
Custom Development Environments	Jean Risley
DEC Professional	Rex Jaeschke
DECUS	Mike Terrazas
Data General	Michael Meissner
	Mark Harris (Alt)
Datapoint	Leonard Ohmes
Data Systems Analysts	James Stanley
Delft Consulting	Chaim Schaap
Digital Equipment Corporation	Randy Meyers
	Art Bjork (Alt)
	Lu Anne Van de Pas (Alt)
Digital Systems International, Inc.	Glen W. Zorn
EDS	Ben Patel
EPI	Richard Relph
Edinburgh Portable Compilers	Graham Andrews
	Colin McPhail (Alt)
Edison Design Group	J. Stephen Adamczyk
	Eric Schwarz (Alt)
Everest Solutions	Dmitry Lenkov
Farance Inc.	Frank Farance
	Peter Hayes (Alt)
Floradin	Florin Jordan
General Electric Information Services	Philip Provin
Gould CSD	Mike Bennett
	Liz Sanville (Alt)
	Tina Aleksa (Alt)
HCR Corporation	Thomas Kelly
	Paul Jackson (Alt)
Harris Computer Systems	Gary Jeter
Hewlett Packard	Sue Meloy
	Walter Murray (Alt)
	Larry Rosler (Alt)
Honeywell Information Systems	Thomas E. Osten
	David Kayden (Alt)
IBM	Shawn Elliott
	Larry Breed (Alt)
	Mel Goldberg (Alt)
Instruction Set	Mike Banahan
Intel	Clark Nelson
	Dan Lau (Alt)
InterACT	John Wolfe
	Lillian Toll (Alt)
Intermetrics	Randy Hudson
International Computers Ltd.	Keith Winter
	Honey M. Schrecker (Alt)
J. Brodie & Associates	Jim Brodie
Kendall Square Research	Jacklin Kotikian
LSI Logic Europe Ltd.	W. Peter Hesse
Language Processors Inc.	John Kaminski
Laurel Arts	David Yost
Lawrence Livermore National Laboratory	Mike Branstetter
Los Alamos National Laboratory	Bob Weaver
Modcomp	Lidia Eberhart
Masscomp	Patricia Jenkins
	Dave Hinman (Alt)
MetaLink	Michael Kearns
MetaWare Incorporated	Tom Pennello
Microsoft	David F. Weil
	Mitch Harder (Alt)
Microware Systems	Kim Kempf
Minnesota Educational Computing	Shane McCarron
Mosaic Technologies	Bruce Olsen

Organization Represented	Name of Representative
Motorola	Michael Paton
NCR	Rick Schubert
	Brian Johnson (Alt)
National Semiconductor	Joseph Mueller
	Derek Godfrey (Alt)
National Bureau of Standards	Jim Upperman
Naval Research Laboratory	James W. Williams
Novell, Inc.	Tom Scribner
	Doug Snapp (Alt)
OCLC	Lisa Simon
Oakland University	Paul Amaranth
Omniware	August R. Hansen
Oracle Complex Systems	Michael Redrow
Oregon Software	Carl Ellis
Perennial	Barry Hedquist
Peritus International	Sassan Hazeghi
	James Holmlund (Alt)
Plum Hall	Thomas Plum
	Christopher Skelly (Alt)
Prime Computer	Andrew Johnson
	Fran Litterio (Alt)
Prismatics	Daniel J. Conrad
Production Languages	David Fritz
Pugh Killeen	Kenneth Pugh
Purdue University	Ed Ramsey
	Stephen Roberts (Alt)
Pyramid Technology	Zona Walcott
	George Basick (Alt)
Quantitative Technology Corp.	Kevin Nolan
	Robert Mueller (Alt)
Que Corporation	Chris DeVoney
Rabbit Software	Jon Tulk
Rational Systems	Terry Colligan
Saber Software Inc.	Samuel C. Kendall
	Stephen Kaufer (Alt)
Saks & Associates	Daniel Saks
	Nancy Saks (Alt)
SAS Institute	Oliver Bradley
	Alan Beale (Alt)
SDRC	Larry Jones
SEI Information Technology	Donald Kossman
SRI International	Kenneth Harrenstien
Sierra Systems	Larry Rosenthal
Southern Bell Telephone	Phil Hempfner
Spruce Technology	Purshotam Rajani
Stellar Computer	Peter Darnell
	Lee W. Cooprider (Alt)
Storage Technology Corp.	Paul Gilmartin
Sun Microsystems	Courtney Meissen
	Alan Fargusson (Alt)
	Steve Muchnick (Alt)
Supercomputer Systems, Inc.	Chuck Rasbold
	Kelly O'Hair (Alt)
Sydetech System Development Technologies, Inc.	Savu Savulescu
Tandem	Henry Richardson
	John M. Hausman (Alt)
Tartan Laboratories	Samuel Harbison
TauMetric	Michael S. Ball
Tektronix	Carl Sutton
	Jim Besemer (Alt)
Texas Instruments	Reid Tatge
Thinking Machines	James Frankel
Tokheim	Ed Brower
	Robert Mansfield (Alt)

Contents

American National Standard
for Information Systems –

Programming Language –
C

1. Introduction

1.1 Purpose

This standard specifies the form and establishes the interpretation of programs written in the C programming language.[1]

5 ### 1.2 Scope

This standard specifies:

- the representation of C programs;

- the syntax and constraints of the C language;

- the semantic rules for interpreting C programs;

10 - the representation of input data to be processed by C programs;

- the representation of output data produced by C programs;

- the restrictions and limits imposed by a conforming implementation of C.

  This standard does not specify:

- the mechanism by which C programs are transformed for use by a data-processing system;

15 - the mechanism by which C programs are invoked for use by a data-processing system;

- the mechanism by which input data are transformed for use by a C program;

- the mechanism by which output data are transformed after being produced by a C program;

- the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;

20 - all minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

---

[1]. This standard is designed to promote the portability of C programs among a variety of data-processing systems. It is intended for use by implementors and knowledgeable programmers, and is not a tutorial. It is accompanied by a Rationale document that explains many of the decisions of the Technical Committee that produced it.

1.3 References

1. "The C Reference Manual" by Dennis M. Ritchie, a version of which was published in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., (1978). Copyright owned by AT&T.

2. *1984 /usr/group Standard* by the /usr/group Standards Committee, Santa Clara, California, USA (November, 1984).

3. ANSI X3/TR-1-82 (1982), *American National Dictionary for Information Processing Systems*, Information Processing Systems Technical Report.

4. ISO 646:1983, *Information Processing — ISO 7-Bit Coded Character Set for Information Interchange*.

5. ANSI/IEEE 754-1985, *American National Standard for Binary Floating-Point Arithmetic*.

6. ISO 4217:1987, *Codes for the Representation of Currencies and Funds*.

1.4 Organization of the Document

This document is divided into four major sections:

1. this introduction;

2. the characteristics of environments that translate and execute C programs;

3. the language syntax, constraints, and semantics;

4. the library facilities.

Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in the section or elsewhere in the standard. References are used to refer to other related sections. A set of appendixes summarizes information contained in the standard. The abstract, the foreword, the examples, the footnotes, the references, and the appendixes are not part of the standard.

1.5 Base Documents

The language section (Section 3) is derived from "The C Reference Manual" by Dennis M. Ritchie, a version of which was published as Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., 1978; copyright owned by AT&T.

The library section (Section 4) is based on the *1984 /usr/group Standard* by the /usr/group Standards Committee, Santa Clara, California, USA (November 14, 1984).

1.6 Definitions of Terms

In this standard, "shall" is to be interpreted as a requirement on an implementation or on a program; conversely, "shall not" is to be interpreted as a prohibition.

The following terms are used in this document:

• Alignment — a requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address.

• Argument — an expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation. Also known as "actual argument" or "actual parameter."

• Bit — the unit of data storage in the execution environment large enough to hold an object that may have one of two values. It need not be possible to express the address of each individual bit of an object.

- Byte — the unit of data storage large enough to hold any member of the basic character set of the execution environment. It shall be possible to express the address of each individual byte of an object uniquely. A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order* bit; the most significant bit is called the *high-order* bit.

- Character — a bit representation that fits in a byte. The representation of each member of the basic character set in both the source and execution environments shall fit in a byte.

- Constraints — syntactic and semantic restrictions by which the exposition of language elements is to be interpreted.

- Diagnostic message — a message belonging to an implementation-defined subset of the implementation's message output.

- Forward references — references to later sections of the standard that contain additional information relevant to this section.

- Implementation — a particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment.

- Implementation-defined behavior — behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.

- Implementation limits — restrictions imposed upon programs by the implementation.

- Locale-specific behavior — behavior that depends on local conventions of nationality, culture, and language that each implementation shall document.

- Multibyte character — a sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set.

- Object — a region of data storage in the execution environment, the contents of which can represent values. Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined. When referenced, an object may be interpreted as having a particular type; see 3.2.2.1.

- Parameter — an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition. Also known as ''formal argument'' or ''formal parameter.''

- Undefined behavior — behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which the standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

  If a ''shall'' or ''shall not'' requirement that appears outside of a constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this standard by the words ''undefined behavior'' or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe ''behavior that is undefined.''

- Unspecified behavior — behavior, for a correct program construct and correct data, for which the standard explicitly imposes no requirements.

Other terms are defined at their first appearance, indicated by *italic* type. Terms explicitly defined in this standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this standard are to be interpreted according to the *American National Dictionary for Information Processing Systems*, Information Processing Systems

5 Technical Report ANSI X3/TR-1-82 (1982).

Examples

An example of unspecified behavior is the order in which the arguments to a function are evaluated.

An example of undefined behavior is the behavior on integer overflow.

10 An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

An example of locale-specific behavior is whether the **islower** function returns true for characters other than the 26 lowercase English letters.

Forward references: bitwise shift operators (3.3.7), expressions (3.3), function calls (3.3.2.2),

15 the **islower** function (4.3.1.6), localization (4.4).

1.7 Compliance

A *strictly conforming program* shall use only those features of the language and library specified in this standard. It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.

20 The two forms of *conforming implementation* are hosted and freestanding. A *conforming hosted implementation* shall accept any strictly conforming program. A *conforming freestanding implementation* shall accept any strictly conforming program in which the use of the features specified in the library section (Section 4) is confined to the contents of the standard headers **<float.h>**, **<limits.h>**, **<stdarg.h>**, and **<stddef.h>**. A conforming implementation

25 may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.[2]

A *conforming program* is one that is acceptable to a conforming implementation.[3]

An implementation shall be accompanied by a document that defines all implementation-defined characteristics and all extensions.

30 **Forward references:** limits **<float.h>** and **<limits.h>** (4.1.4), variable arguments **<stdarg.h>** (4.8), common definitions **<stddef.h>** (4.1.5).

---

2. This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this standard.

3. Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs may depend upon nonportable features of a conforming implementation.

1.8 Future Directions

With the introduction of new devices and extended character sets, new features may be added to the standard. Subsections in the language and library sections warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.

5   Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of the standard. They are retained in the standard because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language or library features) is discouraged.

Forward references: future language directions (3.9.9), future library directions (4.13).

2. Environment

An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the *translation environment* and the *execution environment* in this standard. Their characteristics define and constrain the results of executing
5 conforming C programs constructed according to the syntactic and semantic rules for conforming implementations.

Forward references: In the environment section (Section 2), only a few of many possible forward references have been noted.

2.1 Conceptual Models

2.1.1 Translation Environment
10

2.1.1.1 Program Structure

A C program need not all be translated at the same time. The text of the program is kept in units called *source files* in this standard. A source file together with all the headers and source files included via the preprocessing directive **#include**, less any source lines skipped by any of
15 the conditional inclusion preprocessing directives, is called a *translation unit*. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an
20 executable program.

Forward references: conditional inclusion (3.8.1), linkages of identifiers (3.1.2.2), source file inclusion (3.8.2).

2.1.1.2 Translation Phases

The precedence among the syntax rules of translation is specified by the following phases.[4]

25 1. Physical source file characters are mapped to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences are replaced by corresponding single-character internal representations.

2. Each instance of a new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines. A source file that is not
30 empty shall end in a new-line character, which shall not be immediately preceded by a backslash character.

3. The source file is decomposed into preprocessing tokens[5] and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or comment. Each comment is replaced by one space character. New-line characters
35 are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

4. Preprocessing directives are executed and macro invocations are expanded. A **#include** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.

---

4. Implementations must behave as if these separate phases occur, even though many are typically folded together in practice.

5. As described in 3.1, the process of dividing a source file's characters into preprocessing tokens is context-dependent. For example, see the handling of **<** within a **#include** preprocessing directive.

5. Each source character set member and escape sequence in character constants and string literals is converted to a member of the execution character set.

6. Adjacent character string literal tokens are concatenated and adjacent wide string literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated.

8. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

Forward references: lexical elements (3.1), preprocessing directives (3.8), trigraph sequences (2.2.1.1).

2.1.1.3 Diagnostics

A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit that contains a violation of any syntax rule or constraint. Diagnostic messages need not be produced in other circumstances.[6]

2.1.2 Execution Environments

Two execution environments are defined: *freestanding* and *hosted*. In both cases, *program startup* occurs when a designated C function is called by the execution environment. All objects in static storage shall be *initialized* (set to their initial values) before program startup. The manner and timing of such initialization are otherwise unspecified. *Program termination* returns control to the execution environment.

Forward references: initialization (3.5.7).

2.1.2.1 Freestanding Environment

In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. There are otherwise no reserved external identifiers. Any library facilities available to a freestanding program are implementation-defined.

The effect of program termination in a freestanding environment is implementation-defined.

2.1.2.2 Hosted Environment

A hosted environment need not be provided, but shall conform to the following specifications if present.

2.1.2.2.1 Program Startup

The function called at program startup is named **main**. The implementation declares no prototype for this function. It can be defined with no parameters:

```
int main(void) { /*...*/ }
```

or with two parameters (referred to here as **argc** and **argv**, though any names may be used, as they are local to the function in which they are declared):

---

6. The intent is that an implementation should identify the nature of, and where possible localize, each violation. Of course, an implementation is free to produce any number of diagnostics as long as a valid program is still correctly translated. An implementation may also successfully translate an invalid program.

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

If they are defined, the parameters to the **main** function shall obey the following constraints:

- The value of **argc** shall be nonnegative.

- **argv[argc]** shall be a null pointer.

5 - If the value of **argc** is greater than zero, the array members **argv[0]** through **argv[argc-1]** inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both
10 uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.

- If the value of **argc** is greater than zero, the string pointed to by **argv[0]** represents the *program name*; **argv[0][0]** shall be the null character if the program name is not available from the host environment. If the value of **argc** is greater than one, the strings pointed to
15 by **argv[1]** through **argv[argc-1]** represent the *program parameters*.

- The parameters **argc** and **argv** and the strings pointed to by the **argv** array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

2.1.2.2.2 Program Execution

20   In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library section (Section 4).

2.1.2.2.3 Program Termination

A return from the initial call to the **main** function is equivalent to calling the **exit** function with the value returned by the **main** function as its argument. If the **main** function executes a
25 return that specifies no value, the termination status returned to the host environment is undefined.

Forward references: definition of terms (4.1.1), the **exit** function (4.10.4.3).

2.1.2.3 Program Execution

The semantic descriptions in this standard describe the behavior of an abstract machine in
30 which issues of optimization are irrelevant.

Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. Evaluation of an expression may produce side effects. At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be
35 complete and no side effects of subsequent evaluations shall have taken place.

In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).

40   When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.

An instance of each object with automatic storage duration is associated with each entry into
45 its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

The least requirements on a conforming implementation are:

- At sequence points, volatile objects are stable in the sense that previous evaluations are complete and subsequent evaluations have not yet occurred.

5
- At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

- The input and output dynamics of interactive devices shall take place as specified in 4.9.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages actually appear prior to a program waiting for input.

10   What constitutes an interactive device is implementation-defined.

More stringent correspondences between abstract and actual semantics may be defined by each implementation.

Examples

An implementation might define a one-to-one correspondence between abstract and actual
15   semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword `volatile` would then be redundant.

Alternatively, an implementation might perform various optimizations within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries. In such an implementation, at the time of each
20   function entry and function return where the calling function and the called function are in different translation units, the values of all externally linked objects and of all objects accessible via pointers therein would agree with the abstract semantics. Furthermore, at the time of each such function entry the values of the parameters of the called function and of all objects accessible via pointers therein would agree with the abstract semantics. In this type of
25   implementation, objects referred to by interrupt service routines activated by the `signal` function would require explicit specification of `volatile` storage, as well as other implementation-defined restrictions.

In executing the fragment

```
      char c1, c2;
30    /*...*/
      c1 = c1 + c2;
```

the "integral promotions" require that the abstract machine promote the value of each variable to `int` size and then add the two `int`s and truncate the sum. Provided the addition of two `char`s can be done without creating an overflow exception, the actual execution need only produce the
35   same result, possibly omitting the promotions.

Similarly, in the fragment

```
      float f1, f2;
      double d;
      /*...*/
40    f1 = f2 * d;
```

the multiplication may be executed using single-precision arithmetic if the implementation can ascertain that the result would be the same as if it were executed using double-precision arithmetic (for example, if `d` were replaced by the constant 2.0, which has type `double`). Alternatively, an operation involving only `int`s or `float`s may be executed using double-
45   precision operations if neither range nor precision is lost thereby.

To illustrate the grouping behavior of expressions, in the following fragment

```
int a, b;
/*...*/
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

5
```
a = (((a + 32760) + b) + 5);
```

due to the associativity and precedence of these operators. Thus, the result of the sum ''(a +
32760)'' is next added to b, and that result is then added to 5 which results in the value
assigned to a. On a machine in which overflows produce an exception and in which the range of
values representable by an **int** is [-32768,+32767], the implementation cannot rewrite this
10 expression as

```
a = ((a + b) + 32765);
```

since if the values for a and b were, respectively, -32754 and -15, the sum a + b would
produce an exception while the original expression would not; nor can the expression be rewritten
either as

15
```
a = ((a + 32765) + b);
```
or
```
a = (a + (b + 32765));
```

since the values for a and b might have been, respectively, 4 and -8 or -17 and 12. However
on a machine in which overflows do not produce an exception and in which the results of
20 overflows are reversible, the above expression statement can be rewritten by the implementation
in any of the above ways because the same result will occur.

The grouping of an expression does not completely determine its evaluation. In the following
fragment

```
#include <stdio.h>
25 int sum;
char *p;
/*...*/
sum = sum * 10 - '0' + (*p++ = getchar());
```

the expression statement is grouped as if it were written as

30
```
sum = (((sum * 10) - '0') + ((*(p++)) = (getchar())));
```

but the actual increment of p can occur at any time between the previous sequence point and the
next sequence point (the ;), and the call to **getchar** can occur at any point prior to the need of
its returned value.

Forward references: compound statement, or block (3.6.2), expressions (3.3), files (4.9.3),
35 sequence points (3.3, 3.6), the **signal** function (4.7), type qualifiers (3.5.3).

2.2 Environmental Considerations

2.2.1 Character Sets

Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written, and the set interpreted in the execution environment. The values
5 of the members of the execution character set are implementation-defined; any additional members beyond those required by this section are locale-specific.

In a character constant or string literal, members of the execution character set shall be represented by corresponding members of the source character set or by *escape sequences* consisting of the backslash \ followed by one or more characters. A byte with all bits set to 0,
10 called the *null character*, shall exist in the basic execution character set; it is used to terminate a character string literal.

Both the basic source and basic execution character sets shall have at least the following members: the 26 uppercase letters of the English alphabet

```
     A  B  C  D  E  F  G  H  I  J  K  L  M
15   N  O  P  Q  R  S  T  U  V  W  X  Y  Z
```

the 26 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
```

the 10 decimal digits

```
20   0  1  2  3  4  5  6  7  8  9
```

the following 29 graphic characters

```
     !  "  #  %  &  '  (  )  *  +  ,  -  .  /  :
     ;  <  =  >  ?  [  \  ]  ^  _  {  |  }  ~
```

the space character, and control characters representing horizontal tab, vertical tab, and form feed.
25 In both the source and execution basic character sets, the value of each character after **0** in the above list of decimal digits shall be one greater than the value of the previous. In source files, there shall be some way of indicating the end of each line of text; this standard treats such an end-of-line indicator as if it were a single new-line character. In the execution character set, there shall be control characters representing alert, backspace, carriage return, and new line. If any
30 other characters are encountered in a source file (except in a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token), the behavior is undefined.

Forward references: character constants (3.1.3.4), preprocessing directives (3.8), string literals (3.1.4), comments (3.1.9).

35 2.2.1.1 Trigraph Sequences

All occurrences in a source file of the following sequences of three characters (called *trigraph sequences*[7]) are replaced with the corresponding single character.

---

7. The trigraph sequences enable the input of characters that are not defined in the Invariant Code Set as described in ISO 646:1983, which is a subset of the seven-bit ASCII code set.

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

5

10   No other trigraph sequences exist. Each ? that does not begin one of the trigraphs listed above is not changed.

Example

The following source line

```
printf("Eh???/n");
```

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

```
printf("Eh?\n");
```

2.2.1.2 Multibyte Characters

The source character set may contain multibyte characters, used to represent members of the extended character set. The execution character set may also contain multibyte characters, which
20   need not have the same encoding as for the source character set. For both character sets, the following shall hold:

- The single-byte characters defined in 2.2.1 shall be present.

- The presence, meaning, and representation of any additional members is locale-specific.

- A multibyte character may have a *state-dependent encoding*, wherein each sequence of
25   multibyte characters begins in an *initial shift state* and enters other implementation-defined *shift states* when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.

30   - A byte with all bits zero shall be interpreted as a null character independent of shift state.

- A byte with all bits zero shall not occur in the second or subsequent bytes of a multibyte character.

For the source character set, the following shall hold:

- A comment, string literal, character constant, or header name shall begin and end in the initial
35   shift state.

- A comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

2.2.2 Character Display Semantics

The *active position* is that location on a display device where the next character output by the **fputc** function would appear. The intent of writing a printable character (as defined by the **isprint** function) to a display device is to display a graphic representation of that character at
5   the active position and then advance the active position to the next position on the current line. The direction of writing is locale-specific. If the active position is at the final position of a line (if there is one), the behavior is unspecified.

Alphabetic escape sequences representing nongraphic characters in the execution character set are intended to produce actions on display devices as follows:

10   **\a** (*alert*) Produces an audible or visible alert. The active position shall not be changed.

\b (*backspace*) Moves the active position to the previous position on the current line. If the active position is at the initial position of a line, the behavior is unspecified.

\f (*form feed*) Moves the active position to the initial position at the start of the next logical page.

15   **\n** (*new line*) Moves the active position to the initial position of the next line.

\r (*carriage return*) Moves the active position to the initial position of the current line.

\t (*horizontal tab*) Moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior is unspecified.

20   **\v** (*vertical tab*) Moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the last defined vertical tabulation position, the behavior is unspecified.

Each of these escape sequences shall produce a unique implementation-defined value which can be stored in a single **char** object. The external representations in a text file need not be
25   identical to the internal representations, and are outside the scope of this standard.

Forward references: the **fputc** function (4.9.7.3), the **isprint** function (4.3.1.7).

2.2.3 Signals and Interrupts

Functions shall be implemented such that they may be interrupted at any time by a signal, or may be called by a signal handler, or both, with no alteration to earlier, but still active,
30   invocations' control flow (after the interruption), function return values, or objects with automatic storage duration. All such objects shall be maintained outside the *function image* (the instructions that comprise the executable representation of a function) on a per-invocation basis.

The functions in the standard library are not guaranteed to be reentrant and may modify objects with static storage duration.

35   ## 2.2.4 Environmental Limits

Both the translation and execution environments constrain the implementation of language translators and libraries. The following summarizes the environmental limits on a conforming implementation.

2.2.4.1 Translation Limits

40   The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:[8]

---

8. Implementations should avoid imposing fixed translation limits whenever possible.

- 15 nesting levels of compound statements, iteration control structures, and selection control structures

- 8 nesting levels of conditional inclusion

5
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration

- 31 nesting levels of parenthesized declarators within a full declarator

- 32 nesting levels of parenthesized expressions within a full expression

- 31 significant initial characters in an internal identifier or a macro name

- 6 significant initial characters in an external identifier

10
- 511 external identifiers in one translation unit

- 127 identifiers with block scope declared in one block

- 1024 macro identifiers simultaneously defined in one translation unit

- 31 parameters in one function definition

- 31 arguments in one function call

15
- 31 parameters in one macro definition

- 31 arguments in one macro invocation

- 509 characters in a logical source line

- 509 characters in a character string literal or wide string literal (after concatenation)

- 32767 bytes in an object (in a hosted environment only)

20
- 8 nesting levels for **#include**d files

- 257 **case** labels for a **switch** statement (excluding those for any nested **switch** statements)

- 127 members in a single structure or union

- 127 enumeration constants in a single enumeration

25
- 15 levels of nested structure or union definitions in a single struct-declaration-list

2.2.4.2 Numerical Limits

A conforming implementation shall document all the limits specified in this section, which shall be specified in the headers **<limits.h>** and **<float.h>**.

2.2.4.2.1 Sizes of Integral Types <limits.h>

30   The values given below shall be replaced by constant expressions suitable for use in **#if** preprocessing directives. Moreover, except for **CHAR_BIT** and **MB_LEN_MAX**, the following shall be 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 implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the 35   same sign.

- number of bits for smallest object that is not a bit-field (byte)
   ```
   CHAR_BIT                         8
   ```

- minimum value for an object of type **signed char**
   ```
   SCHAR_MIN                      -127
   ```

40
- maximum value for an object of type **signed char**
   ```
   SCHAR_MAX                      +127
   ```

- maximum value for an object of type **unsigned char**
  UCHAR_MAX   255

- minimum value for an object of type **char**
  CHAR_MIN   *see below*

5 - maximum value for an object of type **char**
  CHAR_MAX   *see below*

- maximum number of bytes in a multibyte character, for any supported locale
  MB_LEN_MAX   1

- minimum value for an object of type **short int**
10   **SHRT_MIN**   −32767

- maximum value for an object of type **short int**
  SHRT_MAX   +32767

- maximum value for an object of type **unsigned short int**
  USHRT_MAX   65535

15 - minimum value for an object of type **int**
  INT_MIN   −32767

- maximum value for an object of type **int**
  INT_MAX   +32767

- maximum value for an object of type **unsigned int**
20   **UINT_MAX**   65535

- minimum value for an object of type **long int**
  LONG_MIN   −2147483647

- maximum value for an object of type **long int**
  LONG_MAX   +2147483647

25 - maximum value for an object of type **unsigned long int**
  ULONG_MAX   4294967295

If the value of an object of type **char** is treated as a signed integer when used in an expression, the value of **CHAR_MIN** shall be the same as that of **SCHAR_MIN** and the value of **CHAR_MAX** shall be the same as that of **SCHAR_MAX**. Otherwise, the value of **CHAR_MIN** shall

30 be 0 and the value of **CHAR_MAX** shall be the same as that of **UCHAR_MAX**.[9]

2.2.4.2.2 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 an implementation's floating-point arithmetic.[10] The following parameters are used to define the

35 model for each floating-point type:

---

9. See 3.1.2.5.

10. The floating-point model is intended to clarify the description of each floating-point characteristic and does not
    require the floating-point arithmetic of the implementation to be identical.

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)

5   f_k nonnegative integers less than b (the significand digits)

A normalized floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} \ , \quad e_{min} \leq e \leq e_{max}$$

Of the values in the **<float.h>** header, **FLT_RADIX** shall be a constant expression suitable for use in **#if** preprocessing directives; all other values need not be constant expressions. All

10 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

15   **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 shall be replaced by implementation-defined expressions

20 that shall be 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

25   **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, $\left\lfloor (p - 1) \times \log_{10}b \right\rfloor + \begin{cases} 1 & \text{if } b \text{ is a power of 10} \\ 0 & \text{otherwise} \end{cases}$

30   **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}

35   **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, $\left\lceil \log_{10}b^{e_{min}-1} \right\rceil$

40   **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, $\left\lfloor \log_{10}((1 - b^{-p}) \times b^{e_{max}}) \right\rfloor$

```
FLT_MAX_10_EXP                          +37
DBL_MAX_10_EXP                          +37
LDBL_MAX_10_EXP                         +37
```

The values given in the following list shall be replaced by implementation-defined expressions with values that shall be 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 shall be replaced by implementation-defined expressions with values that shall be 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 point 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**:

$$x = s \times 16^e \times \sum_{k=1}^{6} f_k \times 16^{-k} , \quad -31 \le e \le +32$$

```
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 ANSI/IEEE 754-1985,[11] and the appropriate values in a **<float.h>** header for types **float** and **double**:

$$x_f = s \times 2^e \times \sum_{k=1}^{24} f_k \times 2^{-k}, \quad -125 \le e \le +128$$

5
$$x_d = s \times 2^e \times \sum_{k=1}^{53} f_k \times 2^{-k}, \quad -1021 \le e \le +1024$$

```
        FLT_RADIX                           2
        FLT_MANT_DIG                       24
        FLT_EPSILON          1.19209290E-07F
        FLT_DIG                            6
10      FLT_MIN_EXP                     -125
        FLT_MIN              1.17549435E-38F
        FLT_MIN_10_EXP                   -37
        FLT_MAX_EXP                     +128
        FLT_MAX              3.40282347E+38F
15      FLT_MAX_10_EXP                   +38
        DBL_MANT_DIG                      53
        DBL_EPSILON 2.2204460492503131E-16
        DBL_DIG                           15
        DBL_MIN_EXP                    -1021
20      DBL_MIN     2.2250738585072014E-308
        DBL_MIN_10_EXP                  -307
        DBL_MAX_EXP                    +1024
        DBL_MAX     1.7976931348623157E+308
        DBL_MAX_10_EXP                  +308
```

25   **Forward references:** conditional inclusion (3.8.1).

---

11. 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.

3. Language

In the syntax notation used in the language section (Section 3), syntactic categories (nonterminals) are indicated by *italic* type, and literal words and character set members (terminals) by **bold** type. A colon (:) following a nonterminal introduces its definition.
5   Alternative definitions are listed on separate lines, except when prefaced by the words "one of." An optional symbol is indicated by the subscript "opt," so that

{ *expression*$_{opt}$ }

indicates an optional expression enclosed in braces.

3.1 Lexical Elements

10   **Syntax**

token:
    keyword
    identifier
    constant
15   *string-literal*
    operator
    punctuator

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

Constraints

Each preprocessing token that is converted to a token shall have the lexical form of a keyword, an identifier, a constant, a string literal, an operator, or a punctuator.

30   **Semantics**

A *token* is the minimal lexical element of the language in translation phases 7 and 8. The categories of tokens are: *keywords, identifiers, constants, string literals, operators,* and *punctuators.* A *preprocessing token* is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: *header names, identifiers,*
35   *preprocessing numbers, character constants, string literals, operators, punctuators,* and single non-white-space characters that do not lexically match the other preprocessing token categories. If a ′ or a " character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by *white space*; this consists of comments (described later), or *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in
40   3.8, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

If the input stream has been parsed into preprocessing tokens up to a given character, the next
45   preprocessing token is the longest sequence of characters that could constitute a preprocessing token.

Examples

The program fragment **1Ex** is parsed as a preprocessing number token (one that is not a valid floating or integer constant token), even though a parse as the pair of preprocessing tokens **1** and **Ex** might produce a valid expression (for example, if **Ex** were a macro defined as **+1**).
5 Similarly, the program fragment **1E1** is parsed as a preprocessing number (one that is a valid floating constant token), whether or not **E** is a macro name.

The program fragment **x+++++y** is parsed as **x ++ ++ + y**, which violates a constraint on increment operators, even though the parse **x ++ + ++ y** might yield a correct expression.

Forward references: character constants (3.1.3.4), comments (3.1.9), expressions (3.3), floating
10 constants (3.1.3.1), header names (3.1.7), macro replacement (3.8.3), postfix increment and decrement operators (3.3.2.4), prefix increment and decrement operators (3.3.3.1), preprocessing directives (3.8), preprocessing numbers (3.1.8), string literals (3.1.4).

3.1.1 Keywords

Syntax

15 *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

Semantics

25 The above tokens (entirely in lowercase) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise.

3.1.2 Identifiers

Syntax

identifier:
30 *nondigit*
identifier nondigit
identifier digit

nondigit: one of

    _ a b c d e f g h i j k l m
35   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

40 **Description**

An identifier is a sequence of nondigit characters (including the underscore _ and the lowercase and uppercase letters) and digits. The first character shall be a nondigit character.

Constraints

In translation phases 7 and 8, an identifier shall not consist of the same sequence of characters
45 as a keyword.

Semantics

An identifier denotes an object, a function, or one of the following entities that will be described later: a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. A member of an enumeration is called an
5 *enumeration constant*. Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.

There is no specific limit on the maximum length of an identifier.

10 **Implementation Limits**

The implementation shall treat at least the first 31 characters of an *internal name* (a macro name or an identifier that does not have external linkage) as significant. Corresponding lowercase and uppercase letters are different. The implementation may further restrict the significance of an *external name* (an identifier that has external linkage) to six characters and may ignore
15 distinctions of alphabetical case for such names.[12] These limitations on identifiers are all implementation-defined.

Any identifiers that differ in a significant character are different identifiers. If two identifiers differ in a nonsignificant character, the behavior is undefined.

Forward references: linkages of identifiers (3.1.2.2), macro replacement (3.8.3).

20 **3.1.2.1 Scopes of Identifiers**

An identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. There are four kinds of scopes: function, file, block, and function prototype. (A *function prototype* is a declaration of a function that declares the types of its parameters.)

A label name is the only kind of identifier that has *function scope*. It can be used (in a **goto**
25 statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a : and a statement). Label names shall be unique within a function.

Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the
30 translation unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the } that closes the associated block. If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which
35 terminates at the end of the function declarator. If an outer declaration of a lexically identical identifier exists in the same name space, it is hidden until the current scope terminates, after which it again becomes visible.

Two identifiers have the same scope if and only if their scopes terminate at the same point.

Structure, union, and enumeration tags have scope that begins just after the appearance of the
40 tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has scope that begins just after the completion of its declarator.

---

12. See "future language directions" (3.9.1).

Forward references: compound statement, or block (3.6.2), declarations (3.5), enumeration specifiers (3.5.2.2), function calls (3.3.2.2), function declarators (including prototypes) (3.5.4.3), function definitions (3.7.1), the **goto** statement (3.6.6.1), labeled statements (3.6.1), name spaces of identifiers (3.1.2.3), scope of macro definitions (3.8.3.5), source file inclusion (3.8.2), tags 5 (3.5.2.3), type specifiers (3.5.2).

3.1.2.2 Linkages of Identifiers

An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*. There are three kinds of linkage: external, internal, and none.

10 In the set of translation units and libraries that constitutes an entire program, each instance of a particular identifier with *external linkage* denotes the same object or function. Within one translation unit, each instance of an identifier with *internal linkage* denotes the same object or function. Identifiers with *no linkage* denote unique entities.

If the declaration of a file scope identifier for an object or a function contains the storage-15 class specifier **static**, the identifier has internal linkage.[13]

If the declaration of an identifier for an object or a function contains the storage-class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage.

20 If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **extern**. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

The following identifiers have no linkage: an identifier declared to be anything other than an 25 object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier **extern**.

If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

Forward references: compound statement, or block (3.6.2), declarations (3.5), expressions (3.3), 30 external definitions (3.7).

3.1.2.3 Name Spaces of Identifiers

If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:

35 • *label names* (disambiguated by the syntax of the label declaration and use);

• the *tags* of structures, unions, and enumerations (disambiguated by following any[14] of the keywords **struct**, **union**, or **enum**);

• the *members* of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the . 40 or **->** operator);

---

13. A function declaration can only contain the storage-class specifier **static** if it is at file scope; see 3.5.1.

14. There is only one name space for tags even though three are possible.

- all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

Forward references: enumeration specifiers (3.5.2.2), labeled statements (3.6.1), structure and union specifiers (3.5.2.1), structure and union members (3.3.2.3), tags (3.5.2.3).

3.1.2.4 Storage Durations of Objects

An object has a *storage duration* that determines its lifetime. There are two storage durations: static and automatic.

An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **static** has *static storage duration*. For such an object, storage is reserved and its stored value is initialized only once, prior to program startup. The object exists and retains its last-stored value throughout the execution of the entire program.[15]

An object whose identifier is declared with no linkage and without the storage-class specifier **static** has *automatic storage duration*. Storage is guaranteed to be reserved for a new instance of such an object on each normal entry into the block with which it is associated, or on a jump from outside the block to a labeled statement in the block or in an enclosed block. If an initialization is specified for the value stored in the object, it is performed on each normal entry, but not if the block is entered by a jump to a labeled statement. Storage for the object is no longer guaranteed to be reserved when execution of the block ends in any way. (Entering an enclosed block suspends but does not end execution of the enclosing block. Calling a function suspends but does not end execution of the block containing the call.) The value of a pointer that referred to an object with automatic storage duration that is no longer guaranteed to be reserved is indeterminate.

Forward references: compound statement, or block (3.6.2), function calls (3.3.2.2), initialization (3.5.7).

3.1.2.5 Types

The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that describe objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes).

An object declared as type **char** is large enough to store any member of the basic execution character set. If a member of the required source character set enumerated in 2.2.1 is stored in a **char** object, its value is guaranteed to be positive. If other quantities are stored in a **char** object, the behavior is implementation-defined: the values are treated as either signed or nonnegative integers.

There are four *signed integer types*, designated as **signed char**, **short int**, **int**, and **long int**. (The signed integer and other types may be designated in several additional ways, as described in 3.5.2.)

An object declared as type **signed char** occupies the same amount of storage as a "plain" **char** object. A "plain" **int** object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range **INT_MIN** to **INT_MAX** as defined in the header **<limits.h>**). In the list of signed integer types above, the range of values of each type is a subrange of the values of the next type in the list.

---

15. In the case of a volatile object, the last store may not be explicit in the program.

For each of the signed integer types, there is a corresponding (but different) *unsigned integer type* (designated with the keyword **unsigned**) that uses the same amount of storage (including sign information) and has the same alignment requirements. The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the
5   representation of the same value in each type is the same.[16] A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type.

There are three *floating types*, designated as **float**, **double**, and **long double**. The set
10   of values of the type **float** is a subset of the set of values of the type **double**; the set of values of the type **double** is a subset of the set of values of the type **long double**.

The type **char**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.

15   The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*.

An *enumeration* comprises a set of named integer constant values. Each distinct enumeration constitutes a different *enumerated type*.

The **void** type comprises an empty set of values; it is an incomplete type that cannot be
20   completed.

Any number of *derived types* can be constructed from the object, function, and incomplete types, as follows:

- An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*.[17] Array types are characterized by their element
25   type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is T, the array type is sometimes called "array of T." The construction of an array type from an element type is called "array type derivation."

- A *structure type* describes a sequentially allocated nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

30   - A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.

- A *function type* describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is T, the function type is
35   sometimes called "function returning T." The construction of a function type from a return type is called "function type derivation."

- A *pointer type* may be derived from a function type, an object type, or an incomplete type, called the *referenced type*. A pointer type describes an object whose value provides a reference to an entity of the referenced type. A pointer type derived from the referenced type
40   T is sometimes called "pointer to T." The construction of a pointer type from a referenced type is called "pointer type derivation."

---

16. The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

17. Since object types do not include incomplete types, an array of incomplete type cannot be constructed.

These methods of constructing derived types can be applied recursively.

The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integral types*. The representations of integral types shall define values by use of a pure binary numeration system.[18] The representations of floating types are unspecified.

5   Integral and floating types are collectively called *arithmetic types*. Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.[19]

An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure
10   or union type of unknown content (as described in 3.5.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.

Array, function, and pointer types are collectively called *derived declarator types*. A *declarator type derivation* from a type T is the construction of a derived declarator type from T
15   by the application of an array-type, a function-type, or a pointer-type derivation to T.

A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.

Any type so far mentioned is an *unqualified type*. Each unqualified type has three
20   corresponding *qualified versions* of its type:[20] a *const-qualified* version, a *volatile-qualified* version, and a version having both qualifications. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.[16] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

25   A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type. Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements.[16] Pointers to other types need not have the same representation or alignment requirements.

Examples

30   The type designated as "**float ***" has type "pointer to **float**." Its type category is pointer, not a floating type. The const-qualified version of this type is designated as "**float * const**" whereas the type designated as "**const float ***" is not a qualified type — its type is "pointer to const-qualified **float**" and is a pointer to a qualified type.

Finally, the type designated as "**struct tag (*[5])(float)**" has type "array of
35   pointer to function returning **struct tag**." The array has length five and the function has a single parameter of type **float**. Its type category is array.

Forward references: character constants (3.1.3.4), compatible type and composite type (3.1.2.6), declarations (3.5), tags (3.5.2.3), type qualifiers (3.5.3).

---

18. A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems*.)

19. Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

20. See 3.5.3 regarding qualified array and function types.

3.1.2.6 Compatible Type and Composite Type

Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described in 3.5.2 for type specifiers, in 3.5.3 for type qualifiers, and in 3.5.4 for declarators.[21] Moreover, two structure, union, or enumeration types
5 declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types; for two structures, the members shall be in the same order; for two structures or unions, the bit-fields shall have the same widths; for two enumerations, the members shall have the same values.

All declarations that refer to the same object or function shall have compatible type;
10 otherwise, the behavior is undefined.

A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

- If one type is an array of known size, the composite type is an array of that size.

- If only one type is a function type with a parameter type list (a function prototype), the
15   composite type is a function prototype with the parameter type list.

- If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

These rules apply recursively to the types from which the two types are derived.

For an identifier with external or internal linkage declared in the same scope as another
20 declaration for that identifier, the type of the identifier becomes the composite type.

Example

Given the following two file scope declarations:

```
int f(int (*)(), double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

25 The resulting composite type for the function is:

```
int f(int (*)(char *), double (*)[3]);
```

Forward references: declarators (3.5.4), enumeration specifiers (3.5.2.2), structure and union specifiers (3.5.2.1), type definitions (3.5.6), type qualifiers (3.5.3), type specifiers (3.5.2).

3.1.3 Constants

30 **Syntax**

constant:
> *floating-constant*
> *integer-constant*
> *enumeration-constant*
35 > *character-constant*

Constraints

The value of a constant shall be in the range of representable values for its type.

---

21. Two types need not be identical to be compatible.

Semantics

Each constant has a type, determined by its form and value, as detailed later.

3.1.3.1 Floating Constants

Syntax

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

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

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

sign: one of
15   **+ −**

digit-sequence:
     digit
     digit-sequence digit

floating-suffix: one of
20   **f l F L**

Description

A floating constant has a *significand part* that may be followed by an *exponent part* and a suffix that specifies its type. The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (**.**), followed by a digit 25 sequence representing the fraction part. The components of the exponent part are an **e** or **E** followed by an exponent consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part shall be present; either the period or the exponent part shall be present.

Semantics

30 The significand part is interpreted as a decimal rational number; the digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of 10 by which the significand part is to be scaled. If the scaled value is in the range of representable values (for its type) the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an 35 implementation-defined manner.

An unsuffixed floating constant has type **double**. If suffixed by the letter **f** or **F**, it has type **float**. If suffixed by the letter **l** or **L**, it has type **long double**.

3.1.3.2 Integer Constants

Syntax

40 *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:
5
>   0
>   *octal-constant octal-digit*

hexadecimal-constant:
>   0x *hexadecimal-digit*
>   0X *hexadecimal-digit*
10
>   *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

15
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:
20
>   *unsigned-suffix long-suffix*$_{opt}$
>   *long-suffix unsigned-suffix*$_{opt}$

unsigned-suffix: one of
>   u   U

long-suffix: one of
25
>   l   L

Description

An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type.

A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits.
30 An octal constant consists of the prefix **0** optionally followed by a sequence of the digits **0** through **7** only. A hexadecimal constant consists of the prefix **0x** or **0X** followed by a sequence of the decimal digits and the letters **a** (or **A**) through **f** (or **F**) with values 10 through 15 respectively.

Semantics

35 The value of a decimal constant is computed base 10; that of an octal constant, base 8; that of a hexadecimal constant, base 16. The lexically first digit is the most significant.

The type of an integer constant is the first of the corresponding list in which its value can be represented. Unsuffixed decimal: **int**, **long int**, **unsigned long int**; unsuffixed octal or hexadecimal: **int**, **unsigned int**, **long int**, **unsigned long int**; suffixed by the letter
40 **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**.

3.1.3.3 Enumeration Constants

Syntax

> *enumeration-constant:*
> > *identifier*

5   **Semantics**

An identifier declared as an enumeration constant has type `int`.

Forward references: enumeration specifiers (3.5.2.2).

3.1.3.4 Character Constants

Syntax

10   *character-constant:*
> > *' c-char-sequence'*
> > **L***' c-char-sequence'*

> *c-char-sequence:*
> > *c-char*
15   > *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*

20   *escape-sequence:*
> > *simple-escape-sequence*
> > *octal-escape-sequence*
> > *hexadecimal-escape-sequence*

> *simple-escape-sequence:* one of
25   > `\'   \"   \?   \\`
> > `\a   \b   \f   \n   \r   \t   \v`

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

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

Description

35   An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in `'x'` or `'ab'`. A wide character constant is the same, except prefixed by the letter **L**. With a few exceptions detailed later, the elements of the sequence are any members of the source character set; they are mapped in an implementation-defined manner to members of the execution character set.

40   The single-quote *'*, the double-quote *"*, the question-mark *?*, the backslash **, and arbitrary integral values, are representable according to the following table of escape sequences:

single-quote '	\'
double-quote "	\"
question-mark ?	\?
backslash \	\\
5 octal integer	*octal digits*
hexadecimal integer	\x*hexadecimal digits*

The double-quote " and question-mark ? are representable either by themselves or by the escape sequences \" and \?, respectively, but the single-quote ' and the backslash \ shall be represented, respectively, by the escape sequences \' and \\.

10   The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the octal integer so formed specifies the value of the desired character or wide character.

The hexadecimal digits that follow the backslash and the letter **x** in a hexadecimal escape 15   sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.

Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

20   In addition, certain nongraphic characters are representable by escape sequences consisting of the backslash \ followed by a lowercase letter: \a, \b, \f, \n, \r, \t, and \v.[22] If any other escape sequence is encountered, the behavior is undefined.[23]

Constraints

The value of an octal or hexadecimal escape sequence shall be in the range of representable 25   values for the type **unsigned char** for an integer character constant, or the unsigned type corresponding to **wchar_t** for a wide character constant.

Semantics

An integer character constant has type **int**. The value of an integer character constant containing a single character that maps into a member of the basic execution character set is the 30   numerical value of the representation of the mapped character interpreted as an integer. The value of an integer character constant containing more than one character, or containing a character or escape sequence not represented in the basic execution character set, is implementation-defined. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **char** whose value is that of 35   the single character or escape sequence is converted to type **int**.

A wide character constant has type **wchar_t**, an integral type defined in the **<stddef.h>** header. The value of a wide character constant containing a single multibyte character that maps into a member of the extended execution character set is the *wide character* (code) corresponding to that multibyte character, as defined by the **mbtowc** function, with an implementation-defined 40   current locale. The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.

---

22. The semantics of these characters were discussed in 2.2.2.

23. See ''future language directions'' (3.9.2).

Examples

The construction '\0' is commonly used to represent the null character.

Consider implementations that use two's-complement representation for integers and eight bits for objects that have type **char**. In an implementation in which type **char** has the same range of values as **signed char**, the integer character constant '\xFF' has the value −1; if type **char** has the same range of values as **unsigned char**, the character constant '\xFF' has the value +255 .

Even if eight bits are used for objects that have type **char**, the construction '\x123' specifies an integer character constant containing only one character. (The value of this single-character integer character constant is implementation-defined and violates the above constraint.) To specify an integer character constant containing the two characters whose values are **0x12** and '3', the construction '\0223' may be used, since a hexadecimal escape sequence is terminated only by a nonhexadecimal character. (The value of this two-character integer character constant is implementation-defined also.)

Even if 12 or more bits are used for objects that have type **wchar_t**, the construction **L**'\1234' specifies the implementation-defined value that results from the combination of the values **0123** and '4'.

Forward references: characters and integers (3.2.1.1) common definitions **<stddef.h>** (4.1.5), the **mbtowc** function (4.10.7.2).

3.1.4 String Literals

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*

Description

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

The same considerations apply to each element of the sequence in a character string literal or a wide string literal as if it were in an integer character constant or a wide character constant, except that the single-quote ' is representable either by itself or by the escape sequence \', but the double-quote " shall be represented by the escape sequence \".

Semantics

In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character string literal tokens, or adjacent wide string literal tokens, are concatenated into a single multibyte character sequence. If a character string literal token is adjacent to a wide string literal token, the behavior is undefined.

In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.[24] The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type **char**, and are initialized
5 with the individual bytes of the multibyte character sequence; for wide string literals, the array elements have type **wchar_t**, and are initialized with the sequence of wide characters corresponding to the multibyte character sequence.

Identical string literals of either form need not be distinct. If the program attempts to modify a string literal of either form, the behavior is undefined.

10 **Example**

This pair of adjacent character string literals

```
"\x12" "3"
```

produces a single character string literal containing the two characters whose values are **\x12** and **'3'**, because escape sequences are converted into single members of the execution character
15 set just prior to adjacent string literal concatenation.

Forward references: common definitions **<stddef.h>** (4.1.5).

3.1.5 Operators

Syntax

operator: one of
20   **[   ]   (   )   .   ->**
              ++   --   &   *   +   -   ~   !   sizeof
              /   %   <<   >>   <   >   <=   >=   ==   !=   ^ | && ||
              ?   :
              =   *=   /=   %=   +=   -=   <<=   >>=   &=   ^= |=
25   **,   #   ##**

Constraints

The operators **[ ]**, **( )**, and **? :** shall occur in pairs, possibly separated by expressions. The operators **#** and **##** shall occur in macro-defining preprocessing directives only.

Semantics

30 An operator specifies an operation to be performed (an *evaluation*) that yields a value, or yields a designator, or produces a side effect, or a combination thereof. An *operand* is an entity on which an operator acts.

Forward references: expressions (3.3), macro replacement (3.8.3).

---

24. A character string literal need not be a string (see 4.1.1), because a null character may be embedded in it by a \0 escape sequence.

3.1.6 Punctuators

Syntax

> *punctuator:* one of
>> [   ]   (   )   {   }   *   ,   :   =   ;   . . .   #

5   **Constraints**

The punctuators **[ ]**, **( )**, and **{ }** shall occur (after translation phase 4) in pairs, possibly separated by expressions, declarations, or statements. The punctuator **#** shall occur in preprocessing directives only.

Semantics

10   A punctuator is a symbol that has independent syntactic and semantic significance but does not specify an operation to be performed that yields a value. Depending on context, the same symbol may also represent an operator or part of an operator.

Forward references: expressions (3.3), declarations (3.5), preprocessing directives (3.8), statements (3.6).

15 ### 3.1.7 Header Names

Syntax

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

20   *h-char-sequence:*
>> *h-char*
>> *h-char-sequence h-char*

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

> *q-char-sequence:*
>> *q-char*
>> *q-char-sequence q-char*

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

Constraints

Header name preprocessing tokens shall only appear within a **#include** preprocessing directive.

35   **Semantics**

The sequences in both forms of header names are mapped in an implementation-defined manner to headers or external source file names as specified in 3.8.2.

If the characters **'**, ****, **"**, or **/*** occur in the sequence between the **<** and **>** delimiters, the behavior is undefined. Similarly, if the characters **'**, ****, or **/*** occur in the sequence between the 40   **"** delimiters, the behavior is undefined.[25]

---

25. Thus, sequences of characters that resemble escape sequences cause undefined behavior.

Example

The following sequence of characters:

```
0x3<1/a.h>1e2
#include <1/a.h>
#define const.member@$
```

forms the following sequence of preprocessing tokens (with each individual preprocessing token delimited by a { on the left and a } on the right).

```
{0x3}{<}{1}{/}{a}{.}{h}{>}{1e2}
{#}{include} {<1/a.h>}
{#}{define} {const}{.}{member}{@}{$}
```

Forward references: source file inclusion (3.8.2).

3.1.8 Preprocessing Numbers

Syntax

pp-number:
       digit
       . *digit*
       pp-number digit
       pp-number nondigit
       pp-number **e** *sign*
       pp-number **E** *sign*
       pp-number .

Description

A preprocessing number begins with a digit optionally preceded by a period (.) and may be followed by letters, underscores, digits, periods, and **e+**, **e-**, **E+**, or **E-** character sequences.

Preprocessing number tokens lexically include all floating and integer constant tokens.

Semantics

A preprocessing number does not have type or a value; it acquires both after a successful conversion (as part of translation phase 7) to a floating constant token or an integer constant token.

3.1.9 Comments

Except within a character constant, a string literal, or a comment, the characters **/*** introduce a comment. The contents of a comment are examined only to identify multibyte characters and to find the characters ***/** that terminate it.[26]

---

26. Thus, comments do not nest.

3.2 Conversions

Several operators convert operand values from one type to another automatically. This section specifies the result required from such an *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*). The list in 3.2.1.5 summarizes the conversions 5 performed by most ordinary operators; it is supplemented as required by the discussion of each operator in 3.3.

Conversion of an operand value to a compatible type causes no change to the value or the representation.

Forward references: cast operators (3.3.4).

10 ### 3.2.1 Arithmetic Operands

3.2.1.1 Characters and Integers

A **char**, a **short int**, or an **int** bit-field, or their signed or unsigned varieties, or an enumeration type, may be used in an expression wherever an **int** or **unsigned int** may be used. If an **int** can represent all values of the original type, the value is converted to an **int**; 15 otherwise, it is converted to an **unsigned int**. These are called the *integral promotions*.[27] All other arithmetic types are unchanged by the integral promotions.

The integral promotions preserve value including sign. As discussed earlier, whether a ''plain'' **char** is treated as signed is implementation-defined.

Forward references: enumeration specifiers (3.5.2.2), structure and union specifiers (3.5.2.1).

20 ### 3.2.1.2 Signed and Unsigned Integers

When a value with integral type is converted to another integral type, if the value can be represented by the new type, its value is unchanged.

When a signed integer is converted to an unsigned integer with equal or greater size, if the value of the signed integer is nonnegative, its value is unchanged. Otherwise: if the unsigned 25 integer has greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned integer; the value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.[28]

When a value with integral type is demoted to an unsigned integer with smaller size, the result is the nonnegative remainder on division by the number one greater than the largest 30 unsigned number that can be represented in the type with smaller size. When a value with integral type is demoted to a signed integer with smaller size, or an unsigned integer is converted to its corresponding signed integer, if the value cannot be represented the result is implementation-defined.

---

27. The integral promotions are applied only as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary +, −, and ~ operators, and to both operands of the shift operators, as specified by their respective sections.

28. In a two's-complement representation, there is no actual change in the bit pattern except filling the high-order bits with copies of the sign bit if the unsigned integer has greater size.

3.2.1.3 Floating and Integral

When a value of floating type is converted to integral type, the fractional part is discarded. If the value of the integral part cannot be represented by the integral type, the behavior is undefined.[29]

5   When a value of integral type is converted to floating type, if the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value, chosen in an implementation-defined manner.

3.2.1.4 Floating Types

When a **float** is promoted to **double** or **long double**, or a **double** is promoted to
10   **long double**, its value is unchanged.

When a **double** is demoted to **float** or a **long double** to **double** or **float**, if the value being converted is outside the range of values that can be represented, the behavior is undefined. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower value,
15   chosen in an implementation-defined manner.

3.2.1.5 Usual Arithmetic Conversions

Many binary operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*:

20   First, if either operand has type **long double**, the other operand is converted to **long double**.

Otherwise, if either operand has type **double**, the other operand is converted to **double**.

Otherwise, if either operand has type **float**, the other operand is converted to **float**.

Otherwise, the integral promotions are performed on both operands. Then the following
25   rules are applied:

If either operand has type **unsigned long int**, the other operand is converted to **unsigned long int**.

Otherwise, if one operand has type **long int** and the other has type **unsigned int**, if a **long int** can represent all values of an **unsigned int**, the operand of
30   type **unsigned int** is converted to **long int**; if a **long int** cannot represent all the values of an **unsigned int**, both operands are converted to **unsigned long int**.

Otherwise, if either operand has type **long int**, the other operand is converted to **long int**.

35   Otherwise, if either operand has type **unsigned int**, the other operand is converted to **unsigned int**.

Otherwise, both operands have type **int**.

The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.[30]

---

29. The remaindering operation performed when a value of integral type is converted to unsigned type need not be performed when a value of floating type is converted to unsigned type. Thus, the range of portable floating values is $(-1, U type\_\mathbf{MAX}+1)$.

30. The cast and assignment operators still must perform their specified conversions, as described in 3.2.1.3 and 3.2.1.4.

3.2.2 Other Operands

3.2.2.1 Lvalues and Function Designators

An *lvalue* is an expression (with an object type or an incomplete type other than **void**) that designates an object.[31] When an object is said to have a particular type, the type is specified by
5 the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member of all contained structures or unions) with a const-qualified type.

Except when it is the operand of the **sizeof** operator, the unary **&** operator, the **++** operator,
10 the **--** operator, or the left operand of the **.** operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue). If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; otherwise, the value has the type of the lvalue. If the lvalue has an incomplete type and does not have array type, the behavior is undefined.

15 Except when it is the operand of the **sizeof** operator or the unary **&** operator, or is a character string literal used to initialize an array of character type, or is a wide string literal used to initialize an array with element type compatible with **wchar_t**, an lvalue that has type ''array of *type*'' is converted to an expression that has type ''pointer to *type*'' that points to the initial element of the array object and is not an lvalue.

20 A *function designator* is an expression that has function type. Except when it is the operand of the **sizeof** operator[32] or the unary **&** operator, a function designator with type ''function returning *type*'' is converted to an expression that has type ''pointer to function returning *type*.''

Forward references: address and indirection operators (3.3.3.2), assignment operators (3.3.16), common definitions **<stddef.h>** (4.1.5), initialization (3.5.7), postfix increment and decrement
25 operators (3.3.2.4), prefix increment and decrement operators (3.3.3.1), the **sizeof** operator (3.3.3.4), structure and union members (3.3.2.3).

3.2.2.2 void

The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to
30 such an expression. If an expression of any other type occurs in a context where a void expression is required, its value or designator is discarded. (A void expression is evaluated for its side effects.)

3.2.2.3 Pointers

A pointer to **void** may be converted to or from a pointer to any incomplete or object type.
35 A pointer to any incomplete or object type may be converted to a pointer to **void** and back again; the result shall compare equal to the original pointer.

For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

---

31. The name ''lvalue'' comes originally from the assignment expression **E1 = E2**, in which the left operand **E1** must be a (modifiable) lvalue. It is perhaps better considered as representing an object ''locator value.'' What is sometimes called ''rvalue'' is in this standard described as the ''value of an expression.''

An obvious example of an lvalue is an identifier of an object. As a further example, if **E** is a unary expression that is a pointer to an object, ***E** is an lvalue that designates the object to which **E** points.

32. Because this conversion does not occur, the operand of the **sizeof** operator remains a function designator and violates the constraint in 3.3.3.4.

An integral constant expression with the value 0, or such an expression cast to type **void ***, is called a *null pointer constant*.[33] If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. Such a pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

5   Two null pointers, converted through possibly different sequences of casts to pointer types, shall compare equal.

Forward references: cast operators (3.3.4), equality operators (3.3.9), simple assignment (3.3.16.1).

---

33. The macro **NULL** is defined in **<stddef.h>** as a null pointer constant; see 4.1.5.

3.3 Expressions

An *expression* is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof.

5   Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be accessed only to determine the value to be stored.[34]

Except as indicated by the syntax[35] or otherwise specified later (for the function-call operator (), &&, ||, ?:, and comma operators), the order of evaluation of subexpressions and the order

10   in which side effects take place are both unspecified.

Some operators (the unary operator ~, and the binary operators <<, >>, &, ^, and |, collectively described as *bitwise operators*) shall have operands that have integral type. These operators return values that depend on the internal representations of integers, and thus have implementation-defined aspects for signed types.

15   If an *exception* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.

An object shall have its stored value accessed only by an lvalue that has one of the following types:[36]

20   • the declared type of the object,

   • a qualified version of the declared type of the object,

   • a type that is the signed or unsigned type corresponding to the declared type of the object,

   • a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,

25   • an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or

   • a character type.

---

34. This paragraph renders undefined statement expressions such as

```
i = ++i + 1;
```

while allowing

```
i = i + 1;
```

35. The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions allowed as the operands of the binary + operator (3.3.6) shall be those expressions defined in 3.3.1 through 3.3.6. The exceptions are cast expressions (3.3.4) as operands of unary operators (3.3.3), and an operand contained between any of the following pairs of operators: grouping parentheses () (3.3.1), subscripting brackets [] (3.3.2.1), function-call parentheses () (3.3.2.2), and the conditional operator ?: (3.3.15).

Within each major subsection, the operators have the same precedence. Left- or right-associativity is indicated in each subsection by the syntax for the expressions discussed therein.

36. The intent of this list is to specify those circumstances in which an object may or may not be aliased.

3.3.1 Primary Expressions

Syntax

> *primary-expression:*
> 5   *identifier*
>   *constant*
>   *string-literal*
>   (*expression*)

Semantics

An identifier is a primary expression, provided it has been declared as designating an object
10 (in which case it is an lvalue) or a function (in which case it is a function designator).

A constant is a primary expression. Its type depends on its form and value, as detailed in
3.1.3.

A string literal is a primary expression. It is an lvalue with type as detailed in 3.1.4.

A parenthesized expression is a primary expression. Its type and value are identical to those
15 of the unparenthesized expression. It is an lvalue, a function designator, or a void expression if
the unparenthesized expression is, respectively, an lvalue, a function designator, or a void
expression.

Forward references: declarations (3.5).

3.3.2 Postfix Operators

20 **Syntax**

> *postfix-expression:*
>   *primary-expression*
>   *postfix-expression* [*expression*]
>   *postfix-expression* (*argument-expression-list*$_{opt}$)
> 25   *postfix-expression* . *identifier*
>   *postfix-expression* **->** *identifier*
>   *postfix-expression* **++**
>   *postfix-expression* **--**
>
> *argument-expression-list:*
> 30   *assignment-expression*
>   *argument-expression-list* , *assignment-expression*

3.3.2.1 Array Subscripting

Constraints

One of the expressions shall have type "pointer to object *type*," the other expression shall
35 have integral type, and the result has type "*type*."

Semantics

A postfix expression followed by an expression in square brackets **[]** is a subscripted
designation of an element of an array object. The definition of the subscript operator **[]** is that
E1[E2] is identical to **(*(E1+(E2)))**. Because of the conversion rules that apply to the
40 binary **+** operator, if **E1** is an array object (equivalently, a pointer to the initial element of an
array object) and **E2** is an integer, **E1[E2]** designates the **E2**-th element of **E1** (counting from
zero).

Successive subscript operators designate an element of a multidimensional array object. If **E**
is an *n*-dimensional array ($n{\geq}2$) with dimensions $i{\times}j{\times} \dots {\times}k$, then **E** (used as other than an
45 lvalue) is converted to a pointer to an ($n-1$)-dimensional array with dimensions $j{\times} \dots {\times}k$. If the
unary ***** operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the

result is the pointed-to ($n-1$)-dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).

Example

5   Consider the array object defined by the declaration

```
int x[3][5];
```

Here **x** is a 3×5 array of **int**s; more precisely, **x** is an array of three element objects, each of which is an array of five **int**s. In the expression **x[i]**, which is equivalent to **(*(x+(i)))**, **x** is first converted to a pointer to the initial array of five **int**s. Then **i** is adjusted according to

10   the type of **x**, which conceptually entails multiplying **i** by the size of the object to which the pointer points, namely an array of five **int** objects. The results are added and indirection is applied to yield an array of five **int**s. When used in the expression **x[i][j]**, that in turn is converted to a pointer to the first of the **int**s, so **x[i][j]** yields an **int**.

Forward references: additive operators (3.3.6), address and indirection operators (3.3.3.2), array

15   declarators (3.5.4.2).

3.3.2.2 Function Calls

Constraints

The expression that denotes the called function[37] shall have type pointer to function returning **void** or returning an object type other than an array type.

20   If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

Semantics

25   A postfix expression followed by parentheses **()** containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.

If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and if no declaration is visible for this identifier, the identifier is implicitly

30   declared exactly as if, in the innermost block containing the function call, the declaration

```
extern int identifier();
```

appeared.[38]

An argument may be an expression of any object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding

35   argument.[39] The value of the function call expression is specified in 3.6.6.4.

---

37. Most often, this is the result of converting an identifier that is a function designator.

38. That is, an identifier with block scope declared to have external linkage with type function without parameter information and returning an **int**. If in fact it is not defined as having type "function returning **int**," the behavior is undefined.

39. A function may change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. A parameter declared to have array or function type is converted to a parameter with a pointer type as described in 3.7.1.

If the expression that denotes the called function has a type that does not include a prototype, the integral promotions are performed on each argument and arguments that have type **float** are promoted to **double**. These are called the *default argument promotions*. If the number of arguments does not agree with the number of parameters, the behavior is undefined. If the
5   function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined. If 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, or if the prototype ends with an ellipsis (, ...), the behavior is undefined.

10 If the expression that denotes the called function has a type that includes a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments. If the function is defined with a type that is not compatible
15 with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.

No other conversions are performed implicitly; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

20 The order of evaluation of the function designator, the arguments, and subexpressions within the arguments is unspecified, but there is a sequence point before the actual call.

Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

Example

25 In the function call

```
(*pf[f1()]) (f2(), f3() + f4())
```

the functions **f1**, **f2**, **f3**, and **f4** may be called in any order. All side effects shall be completed before the function pointed to by **pf[f1()]** is entered.

Forward references: function declarators (including prototypes) (3.5.4.3), function definitions
30 (3.7.1), the **return** statement (3.6.6.4), simple assignment (3.3.16.1).

3.3.2.3 Structure and Union Members

Constraints

The first operand of the . operator shall have a qualified or unqualified structure or union type, and the second operand shall name a member of that type.

35 The first operand of the -> operator shall have type ''pointer to qualified or unqualified structure'' or ''pointer to qualified or unqualified union,'' and the second operand shall name a member of the type pointed to.

Semantics

A postfix expression followed by a dot . and an identifier designates a member of a structure
40 or union object. The value is that of the named member, and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.

A postfix expression followed by an arrow -> and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points, and is an lvalue.[40] If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member.

5   With one exception, if a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined.[41] One special guarantee is made in order to simplify the use of unions: If a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them. Two
10 structures share a *common initial sequence* if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

Examples

If **f** is a function returning a structure or union, and **x** is a member of that structure or union, **f().x** is a valid postfix expression but is not an lvalue.

15 The following is a valid fragment:

```
union {
        struct {
                int     alltypes;
        } n;
        struct {
                int     type;
                int     intnode;
        } ni;
        struct {
                int     type;
                double  doublenode;
        } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
/*...*/
if (u.n.alltypes == 1)
        /*...*/ sin(u.nf.doublenode) /*...*/
```

Forward references: address and indirection operators (3.3.3.2), structure and union specifiers
35 (3.5.2.1).

3.3.2.4 Postfix Increment and Decrement Operators

Constraints

The operand of the postfix increment or decrement operator shall have qualified or unqualified scalar type and shall be a modifiable lvalue.

---

40. If **&E** is a valid pointer expression (where **&** is the "address-of" operator, which generates a pointer to its operand), the expression **(&E)->MOS** is the same as **E.MOS**.

41. The "byte orders" for scalar types are invisible to isolated programs that do not indulge in type punning (for example, by assigning to one member of a union and inspecting the storage by accessing another member that is an appropriately sized array of character type), but must be accounted for when conforming to externally imposed storage layouts.

Semantics

The result of the postfix **++** operator is the value of the operand. After the result is obtained, the value of the operand is incremented. (That is, the value 1 of the appropriate type is added to it.) See the discussions of additive operators and compound assignment for information on
5 constraints, types, and conversions and the effects of operations on pointers. The side effect of updating the stored value of the operand shall occur between the previous and the next sequence point.

The postfix **--** operator is analogous to the postfix **++** operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it).

10 **Forward references:** additive operators (3.3.6), compound assignment (3.3.16.2).

3.3.3 Unary Operators

Syntax

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

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

3.3.3.1 Prefix Increment and Decrement Operators

Constraints

The operand of the prefix increment or decrement operator shall have qualified or unqualified
25 scalar type and shall be a modifiable lvalue.

Semantics

The value of the operand of the prefix **++** operator is incremented. The result is the new value of the operand after incrementation. The expression **++E** is equivalent to **(E+=1)**. See the discussions of additive operators and compound assignment for information on constraints,
30 types, side effects, and conversions and the effects of operations on pointers.

The prefix **--** operator is analogous to the prefix **++** operator, except that the value of the operand is decremented.

Forward references: additive operators (3.3.6), compound assignment (3.3.16.2).

3.3.3.2 Address and Indirection Operators

35 **Constraints**

The operand of the unary **&** operator shall be either a function designator or an lvalue that designates an object that is not a bit-field and is not declared with the **register** storage-class specifier.

The operand of the unary ***** operator shall have pointer type.

40 **Semantics**

The result of the unary **&** (address-of) operator is a pointer to the object or function designated by its operand. If the operand has type ''*type*,'' the result has type ''pointer to *type*.''

The unary ***** operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the

operand has type "pointer to *type*," the result has type "*type*." If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined.[42]

Forward references: storage-class specifiers (3.5.1), structure and union specifiers (3.5.2.1).

3.3.3.3 Unary Arithmetic Operators

5 **Constraints**

The operand of the unary + or − operator shall have arithmetic type; of the ~ operator, integral type; of the ! operator, scalar type.

Semantics

The result of the unary + operator is the value of its operand. The integral promotion is 10 performed on the operand, and the result has the promoted type.

The result of the unary − operator is the negative of its operand. The integral promotion is performed on the operand, and the result has the promoted type.

The result of the ~ operator is the bitwise complement of its operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set). The integral 15 promotion is performed on the operand, and the result has the promoted type. The expression ~E is equivalent to **(ULONG_MAX−E)** if **E** is promoted to type **unsigned long**, to **(UINT_MAX−E)** if **E** is promoted to type **unsigned int**. (The constants **ULONG_MAX** and **UINT_MAX** are defined in the header **<limits.h>**.)

The result of the logical negation operator ! is 0 if the value of its operand compares unequal 20 to 0, 1 if the value of its operand compares equal to 0. The result has type **int**. The expression !E is equivalent to **(0==E)**.

Forward references: limits **<float.h>** and **<limits.h>** (4.1.4).

3.3.3.4 The sizeof Operator

Constraints

25 The **sizeof** operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an lvalue that designates a bit-field object.

Semantics

The **sizeof** operator yields the size (in bytes) of its operand, which may be an expression 30 or the parenthesized name of a type. The size is determined from the type of the operand, which is not itself evaluated. The result is an integer constant.

When applied to an operand that has type **char**, **unsigned char**, or **signed char**, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.[43] When applied to an operand that has structure 35 or union type, the result is the total number of bytes in such an object, including internal and trailing padding.

---

42. It is always true that if **E** is a function designator or an lvalue that is a valid operand of the unary & operator, *&E is a function designator or an lvalue equal to **E**. If *P is an lvalue and **T** is the name of an object pointer type, the cast expression *(T)P is an lvalue that has a type compatible with that to which **T** points.

Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object that has automatic storage duration when execution of the block with which the object is associated has terminated.

43. When applied to a parameter declared to have array or function type, the **sizeof** operator yields the size of the pointer obtained by converting as in 3.2.2.1; see 3.7.1.

The value of the result is implementation-defined, and its type (an unsigned integral type) is **size_t** defined in the **<stddef.h>** header.

Examples

A principal use of the **sizeof** operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to **void**. For example:

```
extern void *alloc(size_t);
double *dp = alloc(sizeof *dp);
```

The implementation of the **alloc** function should ensure that its return value is aligned suitably for conversion to a pointer to **double**.

Another use of the **sizeof** operator is to compute the number of elements in an array:

```
sizeof array / sizeof array[0]
```

Forward references: common definitions **<stddef.h>** (4.1.5), declarations (3.5), structure and union specifiers (3.5.2.1), type names (3.5.5).

3.3.4 Cast Operators

Syntax

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

Constraints

Unless the type name specifies void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.

Semantics

Preceding an expression by a parenthesized type name converts the value of the expression to the named type. This construction is called a *cast*.[44] A cast that specifies no conversion has no effect on the type or value of an expression.

Conversions that involve pointers (other than as permitted by the constraints of 3.3.16.1) shall be specified by means of an explicit cast; they have implementation-defined and undefined aspects:

A pointer may be converted to an integral type. The size of integer required and the result are implementation-defined. If the space provided is not long enough, the behavior is undefined.

An arbitrary integer may be converted to a pointer. The result is implementation-defined.[45]

A pointer to an object or incomplete type may be converted to a pointer to a different object type or a different incomplete type. The resulting pointer might not be valid if it is improperly aligned for the type pointed to. It is guaranteed, however, that a pointer to an object of a given alignment may be converted to a pointer to an object of the same

---

44. A cast does not yield an lvalue. Thus, a cast to a qualified type has the same effect as a cast to the unqualified version of the type.

45. The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment.

alignment or a less strict alignment and back again; the result shall compare equal to the original pointer. (An object that has character type has the least strict alignment.)

A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function that has a type that is not compatible with the type of the called function, the behavior is undefined.

Forward references: equality operators (3.3.9), function declarators (including prototypes) (3.5.4.3), simple assignment (3.3.16.1), type names (3.5.5).

3.3.5 Multiplicative Operators

Syntax

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

Constraints

Each of the operands shall have arithmetic type. The operands of the **%** operator shall have integral type.

Semantics

The usual arithmetic conversions are performed on the operands.

The result of the binary ***** operator is the product of the operands.

The result of the **/** operator is the quotient from the division of the first operand by the second; the result of the **%** operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.

When integers are divided and the division is inexact, if both operands are positive the result of the **/** operator is the largest integer less than the algebraic quotient and the result of the **%** operator is positive. If either operand is negative, whether the result of the **/** operator is the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient is implementation-defined, as is the sign of the result of the **%** operator. If the quotient **a/b** is representable, the expression **(a/b)*b + a%b** shall equal **a**.

3.3.6 Additive Operators

Syntax

> *additive-expression:*
> > *multiplicative-expression*
> > *additive-expression* **+** *multiplicative-expression*
> > *additive-expression* **−** *multiplicative-expression*

Constraints

For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to an object type and the other shall have integral type. (Incrementing is equivalent to adding 1.)

For subtraction, one of the following shall hold:

• both operands have arithmetic type;

• both operands are pointers to qualified or unqualified versions of compatible object types; or

- the left operand is a pointer to an object type and the right operand has integral type. (Decrementing is equivalent to subtracting 1.)

Semantics

If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

The result of the binary **+** operator is the sum of the operands.

The result of the binary **−** operator is the difference resulting from the subtraction of the second operand from the first.

For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

When an expression that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integral expression. In other words, if the expression **P** points to the i-th element of an array object, the expressions **(P)+N** (equivalently, **N+(P)**) and **(P)−N** (where **N** has the value n) point to, respectively, the $i+n$-th and $i-n$-th elements of the array object, provided they exist. Moreover, if the expression **P** points to the last element of an array object, the expression **(P)+1** points one past the last element of the array object, and if the expression **Q** points one past the last element of an array object, the expression **(Q)−1** points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. Unless both the pointer operand and the result point to elements of the same array object, or the pointer operand points one past the last element of an array object and the result points to an element of the same array object, the behavior is undefined if the result is used as an operand of the unary ***** operator.

When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integral type) is **ptrdiff_t** defined in the **<stddef.h>** header. As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined. In other words, if the expressions **P** and **Q** point to, respectively, the i-th and j-th elements of an array object, the expression **(P)−(Q)** has the value $i-j$ provided the value fits in an object of type **ptrdiff_t**. Moreover, if the expression **P** points either to an element of an array object or one past the last element of an array object, and the expression **Q** points to the last element of the same array object, the expression **((Q)+1)−(P)** has the same value as **((Q)−(P))+1** and as **−((P)−((Q)+1))**, and has the value zero if the expression **P** points one past the last element of the array object, even though the expression **(Q)+1** does not point to an element of the array object. Unless both pointers point to elements of the same array object, or one past the last element of the array object, the behavior is undefined.[46]

---

46. Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integral expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which may overlap another object in the program) just after the end of the object in order to satisfy the "one past the last element" requirements.

3.3.7 Bitwise Shift Operators

Syntax

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

Constraints

Each of the operands shall have integral type.

Semantics

The integral promotions are performed on each of the operands. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width in bits of the promoted left operand, the behavior is undefined.

The result of **E1 << E2** is **E1** left-shifted **E2** bit positions; vacated bits are filled with zeros. If **E1** has an unsigned type, the value of the result is **E1** multiplied by the quantity, 2 raised to the power **E2**, reduced modulo **ULONG_MAX+1** if **E1** has type **unsigned long**, **UINT_MAX+1** otherwise. (The constants **ULONG_MAX** and **UINT_MAX** are defined in the header **<limits.h>**.)

The result of **E1 >> E2** is **E1** right-shifted **E2** bit positions. If **E1** has an unsigned type or if **E1** has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of **E1** divided by the quantity, 2 raised to the power **E2**. If **E1** has a signed type and a negative value, the resulting value is implementation-defined.

3.3.8 Relational Operators

Syntax

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

Constraints

One of the following shall hold:

• both operands have arithmetic type;

• both operands are pointers to qualified or unqualified versions of compatible object types; or

• both operands are pointers to qualified or unqualified versions of compatible incomplete types.

Semantics

If both of the operands have arithmetic type, the usual arithmetic conversions are performed.

For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare higher than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values

compare higher than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the objects pointed to are not members of the same aggregate or union object, the result is undefined, with the following exception. If the expression **P** points to an element of an array object and the expression **Q**
5 points to the last element of the same array object, the pointer expression **Q+1** compares higher than **P**, even though **Q+1** does not point to an element of the array object.

If two pointers to object or incomplete types both point to the same object, or both point one past the last element of the same array object, they compare equal. If two pointers to object or incomplete types compare equal, both point to the same object, or both point one past the last
10 element of the same array object.[47]

Each of the operators **<** (less than), **>** (greater than), **<=** (less than or equal to), and **>=** (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.[48] The result has type **int**.

3.3.9 Equality Operators

15 **Syntax**

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

20 **Constraints**

One of the following shall hold:

* both operands have arithmetic type;

* both operands are pointers to qualified or unqualified versions of compatible types;

* one operand is a pointer to an object or incomplete type and the other is a pointer to a
25 qualified or unqualified version of **void**; or

* one operand is a pointer and the other is a null pointer constant.

Semantics

The **==** (equal to) and the **!=** (not equal to) operators are analogous to the relational operators except for their lower precedence.[49] Where the operands have types and values suitable
30 for the relational operators, the semantics detailed in 3.3.8 apply.

If two pointers to object or incomplete types are both null pointers, they compare equal. If two pointers to object or incomplete types compare equal, they both are null pointers, or both point to the same object, or both point one past the last element of the same array object. If two pointers to function types are both null pointers or both point to the same function, they compare
35 equal. If two pointers to function types compare equal, either both are null pointers, or both point to the same function. If one of the operands is a pointer to an object or incomplete type and the other has type pointer to a qualified or unqualified version of **void**, the pointer to an object or incomplete type is converted to the type of the other operand.

---

47. If invalid prior pointer operations, such as accesses outside array bounds, produced undefined behavior, the effect of subsequent comparisons is undefined.

48. The expression **a<b<c** is not interpreted as in ordinary mathematics. As the syntax indicates, it means **(a<b)<c**; in other words, ''if **a** is less than **b** compare 1 to **c**; otherwise, compare 0 to **c**.''

49. Because of the precedences, **a<b == c<d** is 1 whenever **a<b** and **c<d** have the same truth-value.

3.3.10 Bitwise AND Operator

Syntax

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

Constraints

Each of the operands shall have integral type.

Semantics

The usual arithmetic conversions are performed on the operands.

The result of the binary **&** operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

3.3.11 Bitwise Exclusive OR Operator

Syntax

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

Constraints

Each of the operands shall have integral type.

Semantics

The usual arithmetic conversions are performed on the operands.

The result of the **^** operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set).

3.3.12 Bitwise Inclusive OR Operator

Syntax

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

Constraints

Each of the operands shall have integral type.

Semantics

The usual arithmetic conversions are performed on the operands.

The result of the **|** operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

3.3.13 Logical AND Operator

Syntax

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

Constraints

Each of the operands shall have scalar type.

Semantics

The **&&** operator shall yield 1 if both of its operands compare unequal to 0; otherwise, it yields 0. The result has type **int**.

Unlike the bitwise binary **&** operator, the **&&** operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares equal to 0, the second operand is not evaluated.

3.3.14 Logical OR Operator

Syntax

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

Constraints

Each of the operands shall have scalar type.

Semantics

The **| |** operator shall yield 1 if either of its operands compare unequal to 0; otherwise, it yields 0. The result has type **int**.

Unlike the bitwise **|** operator, the **| |** operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares unequal to 0, the second operand is not evaluated.

3.3.15 Conditional Operator

Syntax

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

Constraints

The first operand shall have scalar type.

One of the following shall hold for the second and third operands:

• 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; or

• 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**.

Semantics

The first operand is evaluated; there is a sequence point after its evaluation. The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the value of the second or third operand (whichever is evaluated) is
5   the result.[50]

If both the second and third operands have arithmetic type, the usual arithmetic conversions are performed to bring them to a common type and the result has that type. If both the operands have structure or union type, the result has that type. If both operands have void type, the result has void type.

10   If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types pointed-to by both operands. Furthermore, if both operands are pointers to compatible types or differently qualified versions of a compatible type, the result has the composite type; if one operand is a null pointer constant, the result has the type of the other operand; otherwise, one
15   operand is a pointer to **void** or a qualified version of **void**, in which case the other operand is converted to type pointer to **void**, and the result has that type.

Examples

The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the
20   two pointers have compatible types.

Given the declarations

```
const void *c_vp;
void *vp;
const int *c_ip;
volatile int *v_ip;
int *ip;
const char *c_cp;
```

the third column in the following table is the common type that is the result of a conditional expression in which the first two columns are the second and third operands (in either order):

```
30      c_vp   c_ip   const void *
        v_ip   0      volatile int *
        c_ip   v_ip   const volatile int *
        vp     c_cp   const void *
        ip     c_ip   const int *
35      vp     ip     void *
```

---

50. A conditional expression does not yield an lvalue.

3.3.16 Assignment Operators

Syntax

assignment-expression:
        conditional-expression
5   *unary-expression assignment-operator assignment-expression*

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

Constraints

An assignment operator shall have a modifiable lvalue as its left operand.

10 **Semantics**

An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand. The
15 side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point.

The order of evaluation of the operands is unspecified.

3.3.16.1 Simple Assignment

Constraints

20   One of the following shall hold:[51]

- the left operand has qualified or unqualified arithmetic type and the right has arithmetic type;

- the left operand has a qualified or unqualified version of a structure or union type compatible with the type of the right;

- both operands are pointers to qualified or unqualified versions of compatible types, and the
25   type pointed to by the left has all the qualifiers of the type pointed to by the right;

- 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**, and the type pointed to by the left has all the qualifiers of the type pointed to by the right; or

- the left operand is a pointer and the right is a null pointer constant.

30 **Semantics**

In *simple assignment* (`=`), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.

If the value being stored in an object is accessed from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have
35 qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.

---

51. The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 3.2.2.1) that changes lvalues to "the value of the expression" which removes any type qualifiers from the type category of the expression.

Example

In the program fragment

```
int f(void);
char c;
/*...*/
/*...*/ ((c = f()) == -1) /*...*/
```

the **int** value returned by the function may be truncated when stored in the **char**, and then converted back to **int** width prior to the comparison. In an implementation in which "plain" **char** has the same range of values as **unsigned char** (and **char** is narrower than **int**), the result of the conversion cannot be negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable **c** should be declared as **int**.

3.3.16.2 Compound Assignment

Constraints

For the operators **+=** and **-=** only, either the left operand shall be a pointer to an object type and the right shall have integral type, or the left operand shall have qualified or unqualified arithmetic type and the right shall have arithmetic type.

For the other operators, each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator.

Semantics

A *compound assignment* of the form **E1** *op*= **E2** differs from the simple assignment expression **E1 = E1** *op* **(E2)** only in that the lvalue **E1** is evaluated only once.

3.3.17 Comma Operator

Syntax

expression:
        assignment-expression
        expression , assignment-expression

Semantics

The left operand of a comma operator is evaluated as a void expression; there is a sequence point after its evaluation. Then the right operand is evaluated; the result has its type and value.[52]

Example

As indicated by the syntax, in contexts where a comma is a punctuator (in lists of arguments to functions and lists of initializers) the comma operator as described in this section cannot appear. On the other hand, it can be used within a parenthesized expression or within the second expression of a conditional operator in such contexts. In the function call

```
f(a, (t=3, t+2), c)
```

the function has three arguments, the second of which has the value 5.

Forward references: initialization (3.5.7).

---

52. A comma operator does not yield an lvalue.

3.4 Constant Expressions

Syntax

> *constant-expression:*
> > *conditional-expression*

5 **Description**

A *constant expression* can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be.

Constraints

Constant expressions shall not contain assignment, increment, decrement, function-call, or
10 comma operators, except when they are contained within the operand of a **sizeof** operator.[53]

Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

Semantics

An expression that evaluates to a constant is required in several contexts.[54] If a floating
15 expression is evaluated in the translation environment, the arithmetic precision and range shall be at least as great as if the expression were being evaluated in the execution environment.

An *integral constant expression* shall have integral type and shall only have operands that are integer constants, enumeration constants, character constants, **sizeof** expressions, and floating constants that are the immediate operands of casts. Cast operators in an integral constant
20 expression shall only convert arithmetic types to integral types, except as part of an operand to the **sizeof** operator.

More latitude is permitted for constant expressions in initializers. Such a constant expression shall evaluate to one of the following:

- an arithmetic constant expression,

25 - a null pointer constant,

- an address constant, or

- an address constant for an object type plus or minus an integral constant expression.

An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, and
30 **sizeof** expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to the **sizeof** operator.

An *address constant* is a pointer to an lvalue designating an object of static storage duration, or to a function designator; it shall be created explicitly, using the unary **&** operator, or implicitly, by the use of an expression of array or function type. The array-subscript **[]** and member-access
35 **.** and **->** operators, the address **&** and indirection ***** unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators.

---

53. The operand of a **sizeof** operator is not evaluated (3.3.3.4), and thus any operator in 3.3 may be used.

54. An integral constant expression must be used to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a **case** constant. Further constraints that apply to the integral constant expressions used in conditional-inclusion preprocessing directives are discussed in 3.8.1.

An implementation may accept other forms of constant expressions.

The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.[55]

Forward references: initialization (3.5.7).

---

55. Thus, in the following initialization,

```
static int i = 2 || 1 / 0;
```

the expression is a valid integral constant expression with value one.

3.5 Declarations

Syntax

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

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

> *init-declarator-list:*
10   > *init-declarator*
> > *init-declarator-list , init-declarator*

> *init-declarator:*
> > *declarator*
> > *declarator = initializer*

15 **Constraints**

A declaration shall declare at least a declarator, a tag, or the members of an enumeration.

If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified in 3.5.2.3.

20   All declarations in the same scope that refer to the same object or function shall specify compatible types.

Semantics

A *declaration* specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for an object or function named by an identifier is a
25 *definition.*[56]

The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The init-declarator-list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared.

30   If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer.

Forward references: declarators (3.5.4), enumeration specifiers (3.5.2.2), initialization (3.5.7), tags (3.5.2.3).

---

56. Function definitions have a different syntax, described in 3.7.1.

3.5.1 Storage-Class Specifiers

Syntax

storage-class-specifier:
       `typedef`
5   `extern`
       `static`
       `auto`
       `register`

Constraints

10   At most, one storage-class specifier may be given in the declaration specifiers in a declaration.[57]

Semantics

The **typedef** specifier is called a ''storage-class specifier'' for syntactic convenience only; it is discussed in 3.5.6. The meanings of the various linkages and storage durations were discussed
15   in 3.1.2.2 and 3.1.2.4.

A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.[58]

The declaration of an identifier for a function that has block scope shall have no explicit
20   storage-class specifier other than **extern**.

Forward references: type definitions (3.5.6).

3.5.2 Type Specifiers

Syntax

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

---

57. See ''future language directions'' (3.9.3).

58. The implementation may treat any **register** declaration simply as an **auto** declaration. However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier **register** may not be computed, either explicitly (by use of the unary **&** operator as discussed in 3.3.3.2) or implicitly (by converting an array name to a pointer as discussed in 3.2.2.1). Thus the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof**.

Constraints

Each list of type specifiers shall be one of the following sets (delimited by commas, when there is more than one set on a line); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers.

5   • `void`

       • `char`

       • `signed char`

       • `unsigned char`

       • `short`, `signed short`, `short int`, or `signed short int`

10   • `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`

15   • `float`

       • `double`

       • `long double`

       • struct-or-union specifier

       • enum-specifier

20   • typedef-name

Semantics

Specifiers for structures, unions, and enumerations are discussed in 3.5.2.1 through 3.5.2.3. Declarations of typedef names are discussed in 3.5.6. The characteristics of the other types are discussed in 3.1.2.5.

25   Each of the above comma-separated sets designates the same type, except that for bit-fields, the type `signed int` (or `signed`) may differ from `int` (or no type specifiers).

Forward references: enumeration specifiers (3.5.2.2), structure and union specifiers (3.5.2.1), tags (3.5.2.3), type definitions (3.5.6).

3.5.2.1 Structure and Union Specifiers

30 **Syntax**

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

> *struct-or-union:*
35 >> `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$_{opt}$
5   *type-qualifier specifier-qualifier-list$_{opt}$*

struct-declarator-list:
        struct-declarator
        struct-declarator-list , struct-declarator

struct-declarator:
10   *declarator*
        declarator$_{opt}$: constant-expression

Constraints

A structure or union shall not contain a member with incomplete or function type. Hence it shall not contain an instance of itself (but may contain a pointer to an instance of itself).

15   The expression that specifies the width of a bit-field shall be an integral constant expression that has nonnegative value that shall not exceed the number of bits in an ordinary object of compatible type. If the value is zero, the declaration shall have no declarator.

Semantics

As discussed in 3.1.2.5, a structure is a type consisting of a sequence of named members, 20 whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of named members, whose storage overlap.

Structure and union specifiers have the same form.

The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit. The struct-declaration-list is a sequence of declarations for the members 25 of the structure or union. If the struct-declaration-list contains no named members, the behavior is undefined. The type is incomplete until after the } that terminates the list.

A member of a structure or union may have any object type. In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a *bit-field*;[59] its width is preceded by a colon.

30   A bit-field shall have a type that is a qualified or unqualified version of one of `int`, `unsigned int`, or `signed int`. Whether the high-order bit position of a (possibly qualified) "plain" `int` bit-field is treated as a sign bit is implementation-defined. A bit-field is interpreted as an integral type consisting of the specified number of bits.

An implementation may allocate any addressable storage unit large enough to hold a bit-field. 35 If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.

40   A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.[60] As a special case of this, a bit-field structure member with a width of 0 indicates that

---

59. The unary `&` (address-of) operator may not be applied to a bit-field object; thus, there are no pointers to or arrays of bit-field objects.

60. An unnamed bit-field structure member is useful for padding to conform to externally imposed layouts.

no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.

5   Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may therefore be unnamed padding within a structure object, but not at its beginning, as necessary to achieve the appropriate alignment.

10   The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.

There may also be unnamed padding at the end of a structure or union, as necessary to
15 achieve the appropriate alignment were the structure or union to be an element of an array.

Forward references: tags (3.5.2.3).

3.5.2.2 Enumeration Specifiers

Syntax

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

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

25 *enumerator:*
            enumeration-constant
            enumeration-constant **=** *constant-expression*

Constraints

The expression that defines the value of an enumeration constant shall be an integral constant
30 expression that has a value representable as an **int**.

Semantics

The identifiers in an enumerator list are declared as constants that have type **int** and may appear wherever such are permitted.[61] An enumerator with **=** defines its enumeration constant as the value of the constant expression. If the first enumerator has no **=**, the value of its
35 enumeration constant is 0. Each subsequent enumerator with no **=** defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant. (The use of enumerators with **=** may produce enumeration constants with values that duplicate other values in the same enumeration.) The enumerators of an enumeration are also known as its members.

40   Each enumerated type shall be compatible with an integer type; the choice of type is implementation-defined.

---

61. Thus, the identifiers of enumeration constants declared in the same scope shall all be distinct from each other and from other identifiers declared in ordinary declarators.

Example

```
enum hue { chartreuse, burgundy, claret=20, winedark };
/*...*/
enum hue col, *cp;
/*...*/
col = claret;
cp = &col;
/*...*/
/*...*/ (*cp != burgundy) /*...*/
```

makes **hue** the tag of an enumeration, and then declares **col** as an object that has that type and **cp** as a pointer to an object that has that type. The enumerated values are in the set { 0, 1, 20, 21 }.

Forward references: tags (3.5.2.3).

3.5.2.3 Tags

Semantics

A type specifier of the form

struct-or-union identifier { *struct-declaration-list* }

or

enum *identifier* { *enumerator-list* }

declares the identifier to be the *tag* of the structure, union, or enumeration specified by the list. The list defines the *structure content*, *union content*, or *enumeration content*. If this declaration of the tag is visible, a subsequent declaration that uses the tag and that omits the bracketed list specifies the declared structure, union, or enumerated type. Subsequent declarations in the same scope shall omit the bracketed list.

If a type specifier of the form

struct-or-union identifier

occurs prior to the declaration that defines the content, the structure or union is an incomplete type.[62] It declares a tag that specifies a type that may be used only when the size of an object of the specified type is not needed.[63] If the type is to be completed, another declaration of the tag in the same scope (but not in an enclosed block, which declares a new type known only within that block) shall define the content. A declaration of the form

struct-or-union identifier ;

specifies a structure or union type and declares a tag, both visible only within the scope in which the declaration occurs. It specifies a new type distinct from any type with the same tag in an enclosing scope (if any).

A type specifier of the form

---

62. A similar construction with **enum** does not exist and is not necessary as there can be no mutual dependencies between the declaration of an enumerated type and any other type.

63. It is not needed, for example, when a typedef name is declared to be a specifier for a structure or union, or when a pointer to or a function returning a structure or union is being declared. (See incomplete types in 3.1.2.5.) The specification shall be complete before such a function is called or defined.

> *struct-or-union* **{** *struct-declaration-list* **}**

or

> **enum** **{** *enumerator-list* **}**

5 specifies a new structure, union, or enumerated type, within the translation unit, that can only be referred to by the declaration of which it is a part.[64]

Examples

This mechanism allows declaration of a self-referential structure.

```
        struct tnode {
               int count;
10             struct tnode *left, *right;
        };
```

specifies a structure that contains an integer and two pointers to objects of the same type. Once this declaration has been given, the declaration

```
        struct tnode s, *sp;
```

15 declares **s** to be an object of the given type and **sp** to be a pointer to an object of the given type. With these declarations, the expression **sp->left** refers to the left **struct tnode** pointer of the object to which **sp** points; the expression **s.right->count** designates the **count** member of the right **struct tnode** pointed to from **s**.

The following alternative formulation uses the **typedef** mechanism:

```
20      typedef struct tnode TNODE;
        struct tnode {
               int count;
               TNODE *left, *right;
        };
25      TNODE s, *sp;
```

To illustrate the use of prior declaration of a tag to specify a pair of mutually referential structures, the declarations

```
        struct s1 { struct s2 *s2p; /*...*/ }; /* D1 */
        struct s2 { struct s1 *s1p; /*...*/ }; /* D2 */
```

30 specify a pair of structures that contain pointers to each other. Note, however, that if **s2** were already declared as a tag in an enclosing scope, the declaration **D1** would refer to *it*, not to the tag **s2** declared in **D2**. To eliminate this context sensitivity, the otherwise vacuous declaration

```
        struct s2;
```

may be inserted ahead of **D1**. This declares a new tag **s2** in the inner scope; the declaration **D2** 35 then completes the specification of the new type.

Forward references: type definitions (3.5.6).

---

64. Of course, when the declaration is of a typedef name, subsequent declarations can make use of the typedef name to declare objects having the specified structure, union, or enumerated type.

3.5.3 Type Qualifiers

Syntax

> *type-qualifier:*
> 5   **const**
>   **volatile**

Constraints

The same type qualifier shall not appear more than once in the same specifier list or qualifier list, either directly or via one or more **typedef**s.

Semantics

10   The properties associated with qualified types are meaningful only for expressions that are lvalues.[65]

If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-
15   qualified type, the behavior is undefined.[66]

An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 2.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with
20   that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.[67] What constitutes an access to an object that has volatile-qualified type is implementation-defined.

If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type. If the specification of a function type includes any type qualifiers,
25   the behavior is undefined.[68]

For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.

Examples

30   An object declared

>   **extern const volatile int real_time_clock;**

may be modifiable by hardware, but cannot be assigned to, incremented, or decremented.

The following declarations and expressions illustrate the behavior when type qualifiers modify an aggregate type:

---

65. The implementation may place a **const** object that is not **volatile** in a read-only region of storage. Moreover, the implementation need not allocate storage for such an object if its address is never used.

66. This applies to those objects that behave as if they were defined with qualified types, even if they are never actually defined as objects in the program (such as an object at a memory-mapped input/output address).

67. A **volatile** declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared shall not be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions.

68. Both of these can only occur through the use of **typedef**s.

```
          const struct s { int mem; } cs = { 1 };
          struct s ncs;   /* the object ncs is modifiable */
          typedef int A[2][3];
          const A a = {{4, 5, 6}, {7, 8, 9}}; /* array of array of const int */
5         int *pi;
          const int *pci;

          ncs = cs;       /* valid */
          cs = ncs;       /* violates modifiable lvalue constraint for = */
          pi = &ncs.mem;  /* valid */
10        pi = &cs.mem;   /* violates type constraints for = */
          pci = &cs.mem;  /* valid */
          pi = a[0];      /* invalid: a[0] has type "const int *" */
```

3.5.4 Declarators

Syntax

15 *declarator:*
        *pointer*_{opt} *direct-declarator*

direct-declarator:
        identifier
        (*declarator*)
20   *direct-declarator* [*constant-expression*_{opt}]
        direct-declarator (*parameter-type-list*)
        direct-declarator (*identifier-list*_{opt})

pointer:
        * *type-qualifier-list*_{opt}
25   * *type-qualifier-list*_{opt} *pointer*

type-qualifier-list:
        type-qualifier
        type-qualifier-list *type-qualifier*

parameter-type-list:
30   *parameter-list*
        parameter-list , ...

parameter-list:
        parameter-declaration
        parameter-list , *parameter-declaration*

35 *parameter-declaration:*
        declaration-specifiers *declarator*
        declaration-specifiers *abstract-declarator*_{opt}

identifier-list:
        identifier
40   *identifier-list* , *identifier*

Semantics

Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.

45 In the following subsections, consider a declaration

```
          T D1
```

where **T** contains the declaration specifiers that specify a type T (such as **int**) and **D1** is a declarator that contains an identifier *ident*. The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation.

If, in the declaration "**T D1**," **D1** has the form

5   *identifier*

then the type specified for *ident* is T.

If, in the declaration "**T D1**," **D1** has the form

          (D)

then *ident* has the type specified by the declaration "**T D**." Thus, a declarator in parentheses is
10   identical to the unparenthesized declarator, but the binding of complex declarators may be altered by parentheses.

Implementation Limits

The implementation shall allow the specification of types that have at least 12 pointer, array, and function declarators (in any valid combinations) modifying an arithmetic, a structure, a union,
15   or an incomplete type, either directly or via one or more **typedef**s.

Forward references: type definitions (3.5.6).

3.5.4.1 Pointer Declarators

Semantics

If, in the declaration "**T D1**," **D1** has the form

20   ***** *type-qualifier-list$_{opt}$* **D**

and the type specified for *ident* in the declaration "**T D**" is "*derived-declarator-type-list T*," then the type specified for *ident* is "*derived-declarator-type-list type-qualifier-list* pointer to T." For each type qualifier in the list, *ident* is a so-qualified pointer.

For two pointer types to be compatible, both shall be identically qualified and both shall be
25   pointers to compatible types.

Examples

The following pair of declarations demonstrates the difference between a "variable pointer to a constant value" and a "constant pointer to a variable value."

```
          const int *ptr_to_constant;
30        int *const constant_ptr;
```

The contents of an object pointed to by **ptr_to_constant** shall not be modified through that pointer, but **ptr_to_constant** itself may be changed to point to another object. Similarly, the contents of the **int** pointed to by **constant_ptr** may be modified, but **constant_ptr** itself shall always point to the same location.

35   The declaration of the constant pointer **constant_ptr** may be clarified by including a definition for the type "pointer to **int**."

```
          typedef int *int_ptr;
          const int_ptr constant_ptr;
```

declares **constant_ptr** as an object that has type "const-qualified pointer to **int**."

3.5.4.2 Array Declarators

Constraints

The expression delimited by [and] (which specifies the size of an array) shall be an integral constant expression that has a value greater than zero.

5   **Semantics**

If, in the declaration ''**T D1**,'' **D1** has the form

D [*constant-expression*$_{opt}$]

and the type specified for *ident* in the declaration ''**T D**'' is ''*derived-declarator-type-list T*,'' then the type specified for *ident* is ''*derived-declarator-type-list* array of *T*.''[69] If the size is not

10   present, the array type is an incomplete type.

For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, they shall have the same value.

Examples

```
float fa[11], *afp[17];
```

15   declares an array of **float** numbers and an array of pointers to **float** numbers.

Note the distinction between the declarations

```
extern int *x;
extern int y[];
```

The first declares **x** to be a pointer to **int**; the second declares **y** to be an array of **int** of

20   unspecified size (an incomplete type), the storage for which is defined elsewhere.

Forward references: function definitions (3.7.1), initialization (3.5.7).

3.5.4.3 Function Declarators (Including Prototypes)

Constraints

A function declarator shall not specify a return type that is a function type or an array type.

25   The only storage-class specifier that shall occur in a parameter declaration is **register**.

An identifier list in a function declarator that is not part of a function definition shall be empty.

Semantics

If, in the declaration ''**T D1**,'' **D1** has the form

30   D (*parameter-type-list*)

or

D (*identifier-list*$_{opt}$)

and the type specified for *ident* in the declaration ''**T D**'' is ''*derived-declarator-type-list T*,'' then the type specified for *ident* is ''*derived-declarator-type-list* function returning *T*.''

35   A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function. If the list terminates with an ellipsis (, . . .), no information about the number or types of the parameters after the comma is supplied.[70] The special case of **void** as the only

---

69. When several ''array of'' specifications are adjacent, a multidimensional array is declared.

70. The macros defined in the **<stdarg.h>** header (4.8) may be used to access arguments that correspond to the ellipsis.

item in the list specifies that the function has no parameters.

In a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.

5   The storage-class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition.

An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a function definition specifies that the function has no 10 parameters. The empty list in a function declarator that is not part of a function definition specifies that no information about the number or types of the parameters is supplied.[71]

For two function types to be compatible, both shall specify compatible return types.[72] Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If 15 one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions. If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) 20 identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier. (For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type, as in 3.7.1. For each parameter declared with qualified type, its type for these 25 comparisons is the unqualified version of its declared type.)

Examples

The declaration

```
int f(void), *fip(), (*pfi)();
```

declares a function **f** with no parameters returning an **int**, a function **fip** with no parameter 30 specification returning a pointer to an **int**, and a pointer **pfi** to a function with no parameter specification returning an **int**. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**, so that the declaration suggests, and the same construction in an expression requires, the calling of a function **fip**, and then using indirection through the pointer result to yield an **int**. In the declarator **(*pfi)()**, the extra parentheses are necessary to 35 indicate that indirection through a pointer to a function yields a function designator, which is then used to call the function; it returns an **int**.

If the declaration occurs outside of any function, the identifiers have file scope and external linkage. If the declaration occurs inside a function, the identifiers of the functions **f** and **fip** have block scope and either internal or external linkage (depending on what file scope 40 declarations for these identifiers are visible), and the identifier of the pointer **pfi** has block scope and no linkage.

Here are two more intricate examples.

---

71. See "future language directions" (3.9.4).

72. If both function types are "old style," parameter types are not compared.

```
int (*apfi[3])(int *x, int *y);
```

declares an array **apfi** of three pointers to functions returning **int**. Each of these functions has two parameters that are pointers to **int**. The identifiers **x** and **y** are declared for descriptive purposes only and go out of scope at the end of the declaration of **apfi**. The declaration

5
```
int (*fpfi(int (*)(long), int))(int, ...);
```

declares a function **fpfi** that returns a pointer to a function returning an **int**. The function **fpfi** has two parameters: a pointer to a function returning an **int** (with one parameter of type **long**), and an **int**. The pointer returned by **fpfi** points to a function that has one **int** parameter and accepts zero or more additional arguments of any type.

10 **Forward references:** function definitions (3.7.1), type names (3.5.5).

3.5.5 Type Names

Syntax

> *type-name:*
> > *specifier-qualifier-list abstract-declarator$_{opt}$*

15 > *abstract-declarator:*
> > *pointer*
> > *pointer$_{opt}$ direct-abstract-declarator*
>
> *direct-abstract-declarator:*
> > (*abstract-declarator*)
20 > > *direct-abstract-declarator$_{opt}$* [*constant-expression$_{opt}$*]
> > *direct-abstract-declarator$_{opt}$* (*parameter-type-list$_{opt}$*)

Semantics

In several contexts, it is desired to specify a type. This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the 25 identifier.[73]

Examples

The constructions

> (a) `int`
> (b) `int *`
30 > (c) `int *[3]`
> (d) `int (*)[3]`
> (e) `int *()`
> (f) `int (*)(void)`
> (g) `int (*const [])(unsigned int, ...)`

35 name respectively the types (a) **int**, (b) pointer to **int**, (c) array of three pointers to **int**, (d) pointer to an array of three **int**s, (e) function with no parameter specification returning a pointer to **int**, (f) pointer to function with no parameters returning an **int**, and (g) array of an unspecified number of constant pointers to functions, each with one parameter that has type **unsigned int** and an unspecified number of other parameters, returning an **int**.

---

73. As indicated by the syntax, empty parentheses in a type name are interpreted as "function with no parameter specification," rather than redundant parentheses around the omitted identifier.

3.5.6 Type Definitions

Syntax

> *typedef-name:*
>> *identifier*

5   **Semantics**

In a declaration whose storage-class specifier is **typedef**, each declarator defines an identifier to be a typedef name that specifies the type specified for the identifier in the way described in 3.5.4. A **typedef** declaration does not introduce a new type, only a synonym for the type so specified. That is, in the following declarations:

10
```
typedef T type_ident;
type_ident D;
```

type_ident is defined as a typedef name with the type specified by the declaration specifiers in **T** (known as *T*), and the identifier in **D** has the type "*derived-declarator-type-list T*" where the *derived-declarator-type-list* is specified by the declarators of **D**. A typedef name shares the
15   same name space as other identifiers declared in ordinary declarators. If the identifier is redeclared in an inner scope or is declared as a member of a structure or union in the same or an inner scope, the type specifiers shall not be omitted in the inner declaration.

Examples

After

20
```
typedef int MILES, KLICKSP();
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP *metricp;
```
25
```
complex x;
complex z, *zp;
```

are all valid declarations. The type of **distance** is **int**, that of **metricp** is "pointer to function with no parameter specification returning **int**," and that of **x** and **z** is the specified structure; **zp** is a pointer to such a structure. The object **distance** has a type compatible with
30   any other **int** object.

After the declarations

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type **t1** and the type pointed to by **tp1** are compatible. Type **t1** is also compatible with type
35   **struct s1**, but not compatible with the types **struct s2**, **t2**, the type pointed to by **tp2**, and **int**.

The following obscure constructions

```
typedef signed int t;
typedef int plain;
```
40
```
struct tag {
        unsigned t:4;
        const t:5;
        plain r:5;
};
```

45   declare a typedef name **t** with type **signed int**, a typedef name **plain** with type **int**, and a structure with three bit-field members, one named **t** that contains values in the range [0,15], an

unnamed const-qualified bit-field which (if it could be accessed) would contain values in at least the range [−15,+15], and one named **r** that contains values in the range [0,31] or values in at least the range [−15,+15]. (The choice of range is implementation-defined.) The first two bit-field declarations differ in that **unsigned** is a type specifier (which forces **t** to be the name of a structure member), while **const** is a type qualifier (which modifies **t** which is still visible as a typedef name). If these declarations are followed in an inner scope by

```
t f(t (t));
long t;
```

then a function **f** is declared with 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**,'' and an identifier **t** with type **long**.

On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the **signal** function specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

Forward references: the **signal** function (4.7.1.1).

3.5.7 Initialization

Syntax

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

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

Constraints

There shall be no more initializers in an initializer list than there are objects to be initialized.

The type of the entity to be initialized shall be an object type or an array of unknown size.

All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions.

If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier.

Semantics

An initializer specifies the initial value stored in an object.

All unnamed structure or union members are ignored during initialization.

If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.[74] If an object that has static storage duration is not initialized explicitly, it is

---

74. Unlike in the base document, any automatic duration object may be initialized.

initialized implicitly as if every member that has arithmetic type were assigned 0 and every member that has pointer type were assigned a null pointer constant.

The initializer for a scalar shall be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression; the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type.

A brace-enclosed initializer for a union object initializes the member that appears first in the declaration list of the union type.

The initializer for a structure or union object that has automatic storage duration either shall be an initializer list as described below, or shall be a single expression that has compatible structure or union type. In the latter case, the initial value of the object is that of the expression.

The rest of this section deals with initializers for objects that have aggregate or union type.

An array of character type may be initialized by a character string literal, optionally enclosed in braces. Successive characters of the character string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.

An array with element type compatible with **wchar_t** may be initialized by a wide string literal, optionally enclosed in braces. Successive codes of the wide string literal (including the terminating zero-valued code if there is room or if the array is of unknown size) initialize the elements of the array.

Otherwise, the initializer for an object that has aggregate type shall be a brace-enclosed list of initializers for the members of the aggregate, written in increasing subscript or member order; and the initializer for an object that has union type shall be a brace-enclosed initializer for the first member of the union.

If the aggregate contains members that are aggregates or unions, or if the first member of a union is an aggregate or union, the rules apply recursively to the subaggregates or contained unions. If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the members of the subaggregate or the first member of the contained union. Otherwise, only enough initializers from the list are taken to account for the members of the subaggregate or the first member of the contained union; any remaining initializers are left to initialize the next member of the aggregate of which the current subaggregate or contained union is a part.

If there are fewer initializers in a brace-enclosed list than there are members of an aggregate, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.

If an array of unknown size is initialized, its size is determined by the number of initializers provided for its elements. At the end of its initializer list, the array no longer has incomplete type.

Examples

The declaration

```
        int x[] = { 1, 3, 5 };
```

defines and initializes **x** as a one-dimensional array object that has three elements, as no size was specified and there are three initializers.

```
        float y[4][3] = {
                { 1, 3, 5 },
                { 2, 4, 6 },
                { 3, 5, 7 },
        };
```

is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of y (the array object $y[0]$), namely $y[0][0]$, $y[0][1]$, and $y[0][2]$. Likewise the next two lines initialize $y[1]$ and $y[2]$. The initializer ends early, so $y[3]$ is initialized with zeros. Precisely the same effect could have been achieved by

```
        float y[4][3] = {
                1, 3, 5, 2, 4, 6, 3, 5, 7
        };
```

The initializer for $y[0]$ does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for $y[1]$ and $y[2]$. Also,

```
        float z[4][3] = {
                { 1 }, { 2 }, { 3 }, { 4 }
        };
```

initializes the first column of z as specified and initializes the rest with zeros.

```
        struct { int a[3], b; } w[] = { { 1 }, 2 };
```

is a definition with an inconsistently bracketed initialization. It defines an array with two element structures: $w[0].a[0]$ is 1 and $w[1].a[0]$ is 2; all the other elements are zero.

The declaration

```
        short q[4][3][2] = {
                { 1 },
                { 2, 3 },
                { 4, 5, 6 }
        };
```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: $q[0][0][0]$ is 1, $q[1][0][0]$ is 2, $q[1][0][1]$ is 3, and 4, 5, and 6 initialize $q[2][0][0]$, $q[2][0][1]$, and $q[2][1][0]$, respectively; all the rest are zero. The initializer for $q[0][0]$ does not begin with a left brace, so up to six items from the current list may be used. There is only one, so the values for the remaining five elements are initialized with zero. Likewise, the initializers for $q[1][0]$ and $q[2][0]$ do not begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates. If there had been more than six items in any of the lists, a diagnostic message would have been issued. The same initialization result could have been achieved by:

```
        short q[4][3][2] = {
                1, 0, 0, 0, 0, 0,
                2, 3, 0, 0, 0, 0,
                4, 5, 6
        };
```

or by:

```
        short q[4][3][2] = {
                {
                        { 1 },
                },
                {
                        { 2, 3 },
                },
                {
                        { 4, 5 },
                        { 6 },
                }
        };
```

in a fully bracketed form.

Note that the fully bracketed and minimally bracketed forms of initialization are, in general, less likely to cause confusion.

One form of initialization that completes array types involves typedef names. Given the declaration

```
        typedef int A[];
```

the declaration

```
        A a = {1, 2}, b = {3, 4, 5};
```

is identical to

```
        int a[] = {1, 2}, b[] = {3, 4, 5};
```

due to the rules for incomplete types.

Finally, the declaration

```
        char s[] = "abc", t[3] = "abc";
```

defines ''plain'' **char** array objects **s** and **t** whose elements are initialized with character string literals. This declaration is identical to

```
        char s[] = { 'a', 'b', 'c', '\0' },
              t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
        char *p = "abc";
```

defines **p** with type ''pointer to **char**'' that is initialized to point to an object with type ''array of **char**'' with length 4 whose elements are initialized with a character string literal. If an attempt is made to use **p** to modify the contents of the array, the behavior is undefined.

Forward references: common definitions **<stddef.h>** (4.1.5).

3.6 Statements

Syntax

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

Semantics

A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.

A *full expression* is an expression that is not part of another expression. Each of the following is a full expression: 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 (optional) expressions of a **for** statement; the (optional) expression in a **return** statement. The end of a full expression is a sequence point.

Forward references: expression and null statements (3.6.3), selection statements (3.6.4), iteration statements (3.6.5), the **return** statement (3.6.6.4).

3.6.1 Labeled Statements

Syntax

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

Constraints

A **case** or **default** label shall appear only in a **switch** statement. Further constraints on such labels are discussed under the **switch** statement.

Semantics

Any statement may be preceded by a prefix that declares an identifier as a label name. Labels in themselves do not alter the flow of control, which continues unimpeded across them.

Forward references: the **goto** statement (3.6.6.1), the **switch** statement (3.6.4.2).

3.6.2 Compound Statement, or Block

Syntax

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

Semantics

A *compound statement* (also called a *block*) allows a set of statements to be grouped into one syntactic unit, which may have its own set of declarations and initializations (as discussed in 3.1.2.4). The initializers of objects that have automatic storage duration are evaluated and the values are stored in the objects in the order their declarators appear in the translation unit.

3.6.3 Expression and Null Statements

Syntax

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

Semantics

The expression in an expression statement is evaluated as a void expression for its side effects.[75]

A *null statement* (consisting of just a semicolon) performs no operations.

Examples

If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit by converting the expression to a void expression by means of a cast:

```
int p(int);
/*...*/
(void)p(0);
```

In the program fragment

```
char *s;
/*...*/
while (*s++ != '\0')
        ;
```

a null statement is used to supply an empty loop body to the iteration statement.

A null statement may also be used to carry a label just before the closing } of a compound statement.

```
while (loop1) {
        /*...*/
        while (loop2) {
                /*...*/
                if (want_out)
                        goto end_loop1;
                /*...*/
        }
        /*...*/
end_loop1: ;
}
```

Forward references: iteration statements (3.6.5).

---

75. Such as assignments, and function calls which have side effects.

3.6.4 Selection Statements

Syntax

> *selection-statement:*
>   **if** **(** *expression* **)** *statement*
> 5   **if** **(** *expression* **)** *statement* **else** *statement*
>   **switch** **(** *expression* **)** *statement*

Semantics

A selection statement selects among a set of statements depending on the value of a controlling expression.

10 ### 3.6.4.1 The **if** Statement

Constraints

The controlling expression of an **if** statement shall have scalar type.

Semantics

In both forms, the first substatement is executed if the expression compares unequal to 0. In 15 the **else** form, the second substatement is executed if the expression compares equal to 0. If the first substatement is reached via a label, the second substatement is not executed.

An **else** is associated with the lexically immediately preceding **else**-less **if** that is in the same block (but not in an enclosed block).

3.6.4.2 The **switch** Statement

20 **Constraints**

The controlling expression of a **switch** statement shall have integral type. The expression of each **case** label shall be an integral constant expression. No two of the **case** constant expressions in the same **switch** statement shall have the same value after conversion. There may be at most one **default** label in a **switch** statement. (Any enclosed **switch** statement 25 may have a **default** label or **case** constant expressions with values that duplicate **case** constant expressions in the enclosing **switch** statement.)

Semantics

A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** 30 label and the values of any **case** labels on or in the switch body. A **case** or **default** label is accessible only within the closest enclosing **switch** statement.

The integral promotions are performed on the controlling expression. The constant expression in each **case** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the 35 statement following the matched **case** label. Otherwise, if there is a **default** label, control jumps to the labeled statement. If no converted **case** constant expression matches and there is no **default** label, no part of the switch body is executed.

Implementation Limits

As discussed previously (2.2.4.1), the implementation may limit the number of **case** values 40 in a **switch** statement.

Example

In the artificial program fragment

```
        switch (expr)
        {
                int i = 4;
                f(i);
5       case 0:
                i = 17;     /* falls through into default code */
        default:
                printf("%d\n", i);
        }
```

10   the object whose identifier is **i** exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the **printf** function will access an indeterminate value. Similarly, the call to the function **f** cannot be reached.

3.6.5 Iteration Statements

15 **Syntax**

> *iteration-statement:*
>   **while** (*expression*) *statement*
>   **do** *statement* **while** (*expression*) ;
>   **for** (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*

20 **Constraints**

The controlling expression of an iteration statement shall have scalar type.

Semantics

An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0.

25 **3.6.5.1 The while Statement**

The evaluation of the controlling expression takes place before each execution of the loop body.

3.6.5.2 The do Statement

The evaluation of the controlling expression takes place after each execution of the loop body.

30 **3.6.5.3 The for Statement**

Except for the behavior of a **continue** statement in the loop body, the statement

> **for** (*expression-1* ; *expression-2* ; *expression-3*) *statement*

and the sequence of statements

> *expression-1* ;
35 **while** (*expression-2*) {
>   *statement*
>   *expression-3* ;
> }

are equivalent.[76]

---

76. Thus, *expression-1* specifies initialization for the loop; *expression-2*, the controlling expression, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; *expression-3* specifies an operation (such as incrementing) that is performed after each iteration.

Both *expression-1* and *expression-3* may be omitted. Each is evaluated as a void expression.
An omitted *expression-2* is replaced by a nonzero constant.

Forward references: the `continue` statement (3.6.6.2).

3.6.6 Jump Statements

5   **Syntax**

jump-statement:

```
        goto identifier ;
        continue ;
        break ;
10      return expression   ;
                         opt
```

Semantics

A jump statement causes an unconditional jump to another place.

3.6.6.1 The `goto` Statement

Constraints

15   The identifier in a `goto` statement shall name a label located somewhere in the enclosing
function.

Semantics

A `goto` statement causes an unconditional jump to the statement prefixed by the named label
in the enclosing function.

20   **Example**

It is sometimes convenient to jump into the middle of a complicated set of statements. The
following outline presents one possible approach to a problem based on these three assumptions:

1.   The general initialization code accesses objects only visible to the current function.

2.   The general initialization code is too large to warrant duplication.

25   3.   The code to determine the next operation must be at the head of the loop. (To allow it to
     be reached by `continue` statements, for example.)

```
        /*...*/
        goto first_time;
        for (;;) {
30              /* determine next operation */
                /*...*/
                if (need to reinitialize) {
                        /* reinitialize-only code */
                        /*...*/
35              first_time:
                        /* general initialization code */
                        /*...*/
                        continue;
                }
40              /* handle other operations */
                /*...*/
        }
```

3.6.6.2 The `continue` Statement

Constraints

A `continue` statement shall appear only in or as a loop body.

Semantics

5   A `continue` statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

```
while (/*...*/) {        do {                  for (/*...*/) {
   /*...*/                  /*...*/                /*...*/
   continue;                continue;              continue;
   /*...*/                  /*...*/                /*...*/
   contin: ;                contin: ;              contin: ;
}                        } while (/*...*/);     }
```

unless the `continue` statement shown is in an enclosed iteration statement (in which case it is
15   interpreted within that statement), it is equivalent to `goto contin;`.[77]

3.6.6.3 The `break` Statement

Constraints

A `break` statement shall appear only in or as a switch body or loop body.

Semantics

20   A `break` statement terminates execution of the smallest enclosing `switch` or iteration statement.

3.6.6.4 The `return` Statement

Constraints

A `return` statement with an expression shall not appear in a function whose return type is
25   `void`.

Semantics

A `return` statement terminates execution of the current function and returns control to its caller. A function may have any number of `return` statements, with and without expressions.

If a `return` statement with an expression is executed, the value of the expression is returned
30   to the caller as the value of the function call expression. If the expression has a type different from that of the function in which it appears, it is converted as if it were assigned to an object of that type.

If a `return` statement without an expression is executed, and the value of the function call is used by the caller, the behavior is undefined. Reaching the `}` that terminates a function is
35   equivalent to executing a `return` statement without an expression.

---

77. Following the `contin:` label is a null statement.

3.7 External Definitions

Syntax

5

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

external-declaration:
    function-definition
    declaration

Constraints

10   The storage-class specifiers **auto** and **register** shall not appear in the declaration specifiers in an external declaration.

There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a **sizeof** operator), there shall be exactly 15   one external definition for the identifier in the translation unit.

Semantics

As discussed in 2.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as "external" because they appear outside any function (and hence have file scope). As discussed in 3.5, a declaration 20   that also causes storage to be reserved for an object or a function named by the identifier is a definition.

An *external definition* is an external declaration that is also a definition of a function or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **sizeof** operator), somewhere in the entire program there shall be exactly 25   one external definition for the identifier; otherwise, there shall be no more than one.[78]

3.7.1 Function Definitions

Syntax

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

30 **Constraints**

The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.[79]

---

78. Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

79. The intent is that the type category in a function definition cannot be inherited from a typedef:

```
typedef int F(void);        /* type F is "function of no arguments returning int" */
F f, g;                     /* f and g both have type compatible with F */
F f { /*...*/ }             /* WRONG: syntax/constraint error */
F g() { /*...*/ }           /* WRONG: declares that g returns a function */
int f(void) { /*...*/ }     /* RIGHT: f has type compatible with F */
int g() { /*...*/ }         /* RIGHT: g has type compatible with F */
F *e(void) { /*...*/ }      /* e returns a pointer to a function */
F *((e))(void) { /*...*/ }  /* same: parentheses irrelevant */
int (*fp)(void);            /* fp points to a function that has type F */
F *Fp;                      /* Fp points to a function that has type F */
```

The return type of a function shall be **void** or an object type other than array.

The storage-class specifier, if any, in the declaration specifiers shall be either **extern** or **static**.

5   If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier (except for the special case of a parameter list consisting of a single parameter of type **void**, in which there shall not be an identifier). No declaration list shall follow.

If the declarator includes an identifier list, each declaration in the declaration list shall have at least one declarator, and those declarators shall declare only identifiers from the identifier list. An identifier declared as a typedef name shall not be redeclared as a parameter. The declarations
10   in the declaration list shall contain no storage-class specifier other than **register** and no initializations.

Semantics

The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. If the declarator includes a parameter type list, the list also
15   specifies the types of all the parameters; such a declarator also serves as a function prototype for later calls to the same function in the same translation unit. If the declarator includes an identifier list,[80] the types of the parameters may be declared in a following declaration list. Any parameter that is not declared has type **int**.

If a function that accepts a variable number of arguments is defined without a parameter type
20   list that ends with the ellipsis notation, the behavior is undefined.

On entry to the function the value of each argument expression shall be converted to the type of its corresponding parameter, as if by assignment to the parameter. Array expressions and function designators as arguments are converted to pointers before the call. A declaration of a parameter as ''array of *type*'' shall be adjusted to ''pointer to *type*,'' and a declaration of a
25   parameter as ''function returning *type*'' shall be adjusted to ''pointer to function returning *type*,'' as in 3.2.2.1. The resulting parameter type shall be an object type.

Each parameter has automatic storage duration. Its identifier is an lvalue.[81] The layout of the storage for parameters is unspecified.

Examples

30
```
        extern int max(int a, int b)
        {
                return a > b ? a : b;
        }
```

Here **extern** is the storage-class specifier and **int** is the type specifier (each of which may be
35   omitted as those are the defaults); **max(int a, int b)** is the function declarator; and

```
        { return a > b ? a : b; }
```

is the function body. The following similar definition uses the identifier-list form for the parameter declarations:

---

80. See ''future language directions'' (3.9.5).

81. A parameter is in effect declared at the head of the compound statement that constitutes the function body, and therefore may not be redeclared in the function body (except in an enclosed block).

```
         extern int max(a, b)
         int a, b;
         {
              return a > b ? a : b;
5        }
```

Here **int a, b;** is the declaration list for the parameters, which may be omitted because those are the defaults. The difference between these two definitions is that the first form acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the second form may not.

10   To pass one function to another, one might say

```
         int f(void);
         /*...*/
         g(f);
```

Note that **f** must be declared explicitly in the calling function, as its appearance in the expression
15 **g(f)** was not followed by **(**. Then the definition of **g** might read

```
         g(int (*funcp)(void))
         {
              /*...*/ (*funcp)() /* or funcp() ... */
         }
```

20 or, equivalently,

```
         g(int func(void))
         {
              /*...*/ func() /* or (*func)() ... */
         }
```

25 ## 3.7.2 External Object Definitions

Semantics

If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier.

A declaration of an identifier for an object that has file scope without an initializer, and
30 without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*. If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0.

35   If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type shall not be an incomplete type.

Examples

```
         int i1 = 1;              /* definition, external linkage */
         static int i2 = 2;       /* definition, internal linkage */
40       extern int i3 = 3;       /* definition, external linkage */
         int i4;                  /* tentative definition, external linkage */
         static int i5;           /* tentative definition, internal linkage */
```

```
        int i1;             /* valid tentative definition, refers to previous */
        int i2;             /* 3.1.2.2 renders undefined, linkage disagreement */
        int i3;             /* valid tentative definition, refers to previous */
        int i4;             /* valid tentative definition, refers to previous */
5       int i5;             /* 3.1.2.2 renders undefined, linkage disagreement */

        extern int i1;      /* refers to previous, whose linkage is external */
        extern int i2;      /* refers to previous, whose linkage is internal */
        extern int i3;      /* refers to previous, whose linkage is external */
        extern int i4;      /* refers to previous, whose linkage is external */
10      extern int i5;      /* refers to previous, whose linkage is internal */
```

3.8 Preprocessing Directives

Syntax

preprocessing-file:
    group$_{opt}$

5   *group:*
    group-part
    group group-part

group-part:
    pp-tokens$_{opt}$ *new-line*
10   *if-section*
    control-line

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

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

elif-groups:
    elif-group
20   *elif-groups elif-group*

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

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

25 *endif-line:*
    # endif *new-line*

control-line:
    # include *pp-tokens new-line*
    # define *identifier replacement-list new-line*
30   **# 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*
35   **#**   *new-line*

lparen:
    the left-parenthesis character without preceding white-space

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

40 *pp-tokens:*
    preprocessing-token
    pp-tokens preprocessing-token

new-line:
    the new-line character

Description

A preprocessing directive consists of a sequence of preprocessing tokens that begins with a **#** preprocessing token that is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line 5 character, and is ended by the next new-line character.[82]

Constraints

The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing **#** preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have 10 replaced comments or possibly other white-space characters in translation phase 3).

Semantics

The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

15 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

3.8.1 Conditional Inclusion

Constraints

The expression that controls conditional inclusion shall be an integral constant expression 20 except that: it shall not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below;[83] and it may contain unary operator expressions of the form

> **defined** *identifier*

or

> **defined (** *identifier* **)**

25 which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive with the same subject identifier), 0 if it is not.

Each preprocessing token that remains after all macro replacements have occurred shall be in the lexical form of a token.

30 **Semantics**

Preprocessing directives of the forms

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

check whether the controlling constant expression evaluates to nonzero.

35 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the **defined** unary operator), just as in normal text. If the token **defined** is generated as a result of this replacement process or use of the **defined** unary operator does not match one of the two

---

82. Thus, preprocessing directives are commonly called ''lines.'' These ''lines'' have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the **#** character string literal creation operator in 3.8.3.2, for example).

83. Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, and so on.

specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the **defined** unary operator have been performed, all remaining identifiers are replaced with the pp-number **0**, and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 3.4 using arithmetic that has at least the ranges specified in 2.2.4.2, except that **int** and **unsigned int** act as if they have the same representation as, respectively, **long** and **unsigned long**. This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a **#if** or **#elif** directive) is implementation-defined.[84] Also, whether a single-character character constant may have a negative value is implementation-defined.

Preprocessing directives of the forms

> # **ifdef** *identifier new-line group*$_{opt}$
> # **ifndef** *identifier new-line group*$_{opt}$

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to **#if defined** *identifier* and **#if !defined** *identifier* respectively.

Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed. If none of the conditions evaluates to true, and there is a **#else** directive, the group controlled by the **#else** is processed; lacking a **#else** directive, all the groups until the **#endif** are skipped.[85]

Forward references: macro replacement (3.8.3), source file inclusion (3.8.2).

3.8.2 Source File Inclusion

Constraints

A **#include** directive shall identify a header or source file that can be processed by the implementation.

Semantics

A preprocessing directive of the form

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

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the **<** and **>** delimiters, and causes the replacement of that directive by the entire contents of the header.* How the places are specified or the header identified is implementation-defined.

---

84. Thus, the constant expression in the following **#if** directive and **if** statement is not guaranteed to evaluate to the same value in these two contexts.

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

85. As indicated by the syntax, a preprocessing token shall not follow a **#else** or **#endif** directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.

A preprocessing directive of the form

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

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

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

with the identical contained sequence (including **>** characters, if any) from the original directive.

A preprocessing directive of the form

> **# include** *pp-tokens* *new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **include** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms.[86] The method by which a sequence of preprocessing tokens between a **<** and a **>** preprocessing token pair or a pair of **"** characters is combined into a single header name preprocessing token is implementation-defined.

There shall be an implementation-defined mapping between the delimited sequence and the external source file name. The implementation shall provide unique mappings for sequences consisting of one or more letters (as defined in 2.2.1) followed by a period (.) and a single letter. The implementation may ignore the distinctions of alphabetical case and restrict the mapping to six significant characters before the period.

A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file, up to an implementation-defined nesting limit (see 2.2.4.1).

Examples

The most common uses of **#include** preprocessing directives are as in the following:

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

This example illustrates a macro-replaced **#include** directive:

```
#if VERSION == 1
        #define INCFILE   "vers1.h"
#elif VERSION == 2
        #define INCFILE   "vers2.h"    /* and so on */
#else
        #define INCFILE   "versN.h"
#endif
#include INCFILE
```

Forward references: macro replacement (3.8.3).

---

86. Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 2.1.1.2); thus, an expansion that results in two string literals is an invalid directive.

3.8.3 Macro Replacement

Constraints

Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

An identifier currently defined as a macro without use of lparen (an *object-like* macro) may be redefined by another **#define** preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical.

An identifier currently defined as a macro using lparen (a *function-like* macro) may be redefined by another **#define** preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.

The number of arguments in an invocation of a function-like macro shall agree with the number of parameters in the macro definition, and there shall exist a **)** preprocessing token that terminates the invocation.

A parameter identifier in a function-like macro shall be uniquely declared within its scope.

Semantics

The identifier immediately following the **define** is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

If a **#** preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

A preprocessing directive of the form

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

defines an object-like macro that causes each subsequent instance of the macro name[87] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive. The replacement list is then rescanned for more macro names as specified below.

A preprocessing directive of the form

> **# define** *identifier lparen identifier-list$_{opt}$* **)** *replacement-list new-line*

defines a function-like macro with arguments, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **#define** preprocessing directive. Each subsequent instance of the function-like macro name followed by a **(** as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching **)** preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

---

87. Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 2.1.1.2, translation phases), they are never scanned for macro names or parameters.

The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the behavior is undefined.

3.8.3.1 Argument Substitution

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the translation unit; no other preprocessing tokens are available.

3.8.3.2 The # Operator

Constraints

Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

Semantics

If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token comprising the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters). If the replacement that results is not a valid character string literal, the behavior is undefined. The order of evaluation of # and ## operators is unspecified.

3.8.3.3 The ## Operator

Constraints

A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

Semantics

If, in the replacement list, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence.

For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

3.8.3.4 Rescanning and Further Replacement

After all parameters in the replacement list have been substituted, the resulting preprocessing token sequence is rescanned with all subsequent preprocessing tokens of the source file for more macro names to replace.

5   If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Further, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token
10   would otherwise have been replaced.

The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one.

3.8.3.5 Scope of Macro Definitions

A macro definition lasts (independent of block structure) until a corresponding **#undef**
15   directive is encountered or (if none is encountered) until the end of the translation unit.

A preprocessing directive of the form

> # **undef** *identifier new-line*

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

20   **Examples**

The simplest use of this facility is to define a ''manifest constant,'' as in

```
#define TABSIZE 100

int table[TABSIZE];
```

The following defines a function-like macro whose value is the maximum of its arguments. It
25   has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b)  ((a) > (b) ? (a) : (b))
```

30   The parentheses ensure that the arguments and the resulting expression are bound properly.

To illustrate the rules for redefinition and reexamination, the sequence

```
#define x     3
#define f(a)  f(x * (a))
#undef  x
#define x     2
#define g     f
#define z     z[0]
#define h     g(~
#define m(a)  a(w)
#define w     0,1
#define t(a)  a

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
        (f)^m(m);
```

45   results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
```

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

```
5      #define str(s)      # s
       #define xstr(s)      str(s)
       #define debug(s, t)  printf("x" # s "= %d, x" # t "= %s", \
                                   x ## s, x ## t)
       #define INCFILE(n)   vers ## n   /* from previous #include example */
10     #define glue(a, b)   a ## b
       #define xglue(a, b)  glue(a, b)
       #define HIGHLOW      "hello"
       #define LOW          LOW ", world"

       debug(1, 2);
15     fputs(str(strncmp("abc\0d", "abc", '\4')  /* this goes away */
             == 0) str(: @\n), s);
       #include xstr(INCFILE(2).h)
       glue(HIGH, LOW);
       xglue(HIGH, LOW)
```

20   results in

```
       printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
       fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n", s);
       #include "vers2.h"      (after macro replacement, before file access)
       "hello";
25     "hello" ", world"
```

or, after concatenation of the character string literals,

```
       printf("x1= %d, x2= %s", x1, x2);
       fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);
       #include "vers2.h"      (after macro replacement, before file access)
30     "hello";
       "hello, world"
```

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

And finally, to demonstrate the redefinition rules, the following sequence is valid.

```
       #define OBJ_LIKE        (1-1)
35     #define OBJ_LIKE        /* white space */ (1-1) /* other */
       #define FTN_LIKE(a)     ( a )
       #define FTN_LIKE( a )(          /* note the white space */ \
                              a /* other stuff on this line
                                */ )
```

40   But the following redefinitions are invalid:

```
       #define OBJ_LIKE        (0)        /* different token sequence */
       #define OBJ_LIKE        (1 - 1) /* different white space */
       #define FTN_LIKE(b)     ( a )      /* different parameter usage */
       #define FTN_LIKE(b) ( b )      /* different parameter spelling */
```

3.8.4 Line Control

Constraints

The string literal of a **#line** directive, if present, shall be a character string literal.

Semantics

5   The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (2.1.1.2) while processing the source file to the current token.

A preprocessing directive of the form

> **# line** *digit-sequence new-line*

10   causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). The digit sequence shall not specify zero, nor a number greater than 32767.

A preprocessing directive of the form

> **# line** *digit-sequence "s-char-sequence$_{opt}$" new-line*

15   sets the line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

A preprocessing directive of the form

> **# line** *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after 20 **line** on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.

3.8.5 Error Directive

25 **Semantics**

A preprocessing directive of the form

> **# error** *pp-tokens$_{opt}$ new-line*

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

30 ### 3.8.6 Pragma Directive

Semantics

A preprocessing directive of the form

> **# pragma** *pp-tokens$_{opt}$ new-line*

causes the implementation to behave in an implementation-defined manner. Any pragma that is 35 not recognized by the implementation is ignored.

3.8.7 Null Directive

Semantics

A preprocessing directive of the form

> **#** *new-line*

5   has no effect.

3.8.8 Predefined Macro Names

The following macro names shall be defined by the implementation:

__LINE__   The line number of the current source line (a decimal constant).

__FILE__   The presumed name of the source file (a character string literal).

10   **__DATE__**   The date of translation of the source file (a character string literal of the form **"Mmm dd yyyy"**, where the names of the months are the same as those generated by the **asctime** function, and the first character of **dd** is a space character if the value is less than 10). If the date of translation is not available, an implementation-defined valid date shall be supplied.

15   **__TIME__**   The time of translation of the source file (a character string literal of the form **"hh:mm:ss"** as in the time generated by the **asctime** function). If the time of translation is not available, an implementation-defined valid time shall be supplied.

__STDC__   The decimal constant 1, intended to indicate a conforming implementation.

The values of the predefined macros (except for **__LINE__** and **__FILE__**) remain
20   constant throughout the translation unit.

None of these macro names, nor the identifier **defined**, shall be the subject of a **#define** or a **#undef** preprocessing directive. All predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

Forward references: the **asctime** function (4.12.3.1).

3.9 Future Language Directions

3.9.1 External Names

Restriction of the significance of an external name to fewer than 31 characters or to only one case is an obsolescent feature that is a concession to existing implementations.

3.9.2 Character Escape Sequences

Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

3.9.3 Storage-Class Specifiers

The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

3.9.4 Function Declarators

The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

3.9.5 Function Definitions

The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.

3.9.6 Array Parameters

The use of two parameters declared with an array type (prior to their adjustment to pointer type) in separate lvalues to designate the same object is an obsolescent feature.

4. Library

4.1 Introduction

4.1.1 Definitions of Terms

5   A *string* is a contiguous sequence of characters terminated by and including the first null character. A "pointer to" a string is a pointer to its initial (lowest addressed) character. The "length" of a string is the number of characters preceding the null character and its "value" is the sequence of the values of the contained characters, in order.

        A *letter* is a printing character in the execution character set corresponding to any of the 52
10   required lowercase and uppercase letters in the source character set, listed in 2.2.1.

        The *decimal-point character* is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences.[88] It is represented in the text and examples by a period, but may be changed by the **setlocale** function.

15   **Forward references:** character handling (4.3), the **setlocale** function (4.4.1.1).

4.1.2 Standard Headers

        Each library function is declared in a *header*,[89] whose contents are made available by the **#include** preprocessing directive. The header declares a set of related functions, plus any necessary types and additional macros needed to facilitate their use.

20   The standard headers are

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

        If a file with the same name as one of the above **<** and **>** delimited sequences, not provided as part of the implementation, is placed in any of the standard places for a source file to be included, the behavior is undefined.

        Headers may be included in any order; each may be included more than once in a given
30   scope, with no effect different from being included only once, except that the effect of including **<assert.h>** depends on the definition of **NDEBUG**. If used, a header shall be included outside of any external declaration or definition, and it shall first be included before the first reference to any of the functions or objects it declares, or to any of the types or macros it defines. However, if the identifier is declared or defined in more than one header, the second and subsequent
35   associated headers may be included after the initial reference to the identifier. The program shall not have any macros with names lexically identical to keywords currently defined prior to the inclusion.

        Forward references: diagnostics (4.2).

---

88. The functions that make use of the decimal-point character are **localeconv**, **fprintf**, **fscanf**, **printf**, **scanf**, **sprintf**, **sscanf**, **vfprintf**, **vprintf**, **vsprintf**, **atof**, and **strtod**.

89. A header is not necessarily a source file, nor are the **<** and **>** delimited sequences in header names necessarily valid source file names.

4.1.2.1 Reserved Identifiers

Each header declares or defines all identifiers listed in its associated section, and optionally declares or defines identifiers listed in its associated future library directions section and identifiers which are always reserved either for any use or for use as file scope identifiers.

5 • All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved for any use.

• All identifiers that begin with an underscore are always reserved for use as identifiers with file scope in both the ordinary identifier and tag name spaces.

• Each macro name listed in any of the following sections (including the future library 10 directions) is reserved for any use if any of its associated headers is included.

• All identifiers with external linkage in any of the following sections (including the future library directions) are always reserved for use as identifiers with external linkage.[90]

• Each identifier with file scope listed in any of the following sections (including the future library directions) is reserved for use as an identifier with file scope in the same name space if 15 any of its associated headers is included.

No other identifiers are reserved. If the program declares or defines an identifier with the same name as an identifier reserved in that context (other than as allowed by 4.1.6), the behavior is undefined.[91]

4.1.3 Errors <errno.h>

20 The header **<errno.h>** defines several macros, all relating to the reporting of error conditions.

The macros are

> **EDOM**
> **ERANGE**

25 which expand to integral constant expressions with distinct nonzero values, suitable for use in **#if** preprocessing directives; and

> **errno**

which expands to a modifiable lvalue[92] that has type **int**, the value of which is set to a positive error number by several library functions. It is unspecified whether **errno** is a macro or an 30 identifier declared with external linkage. If a macro definition is suppressed in order to access an actual object, or a program defines an identifier with the name **errno**, the behavior is undefined.

The value of **errno** is zero at program startup, but is never set to zero by any library function.[93] The value of **errno** may be set to nonzero by a library function call whether or not there is an error, provided the use of **errno** is not documented in the description of the function 35 in the standard.

---

90. The list of reserved identifiers with external linkage includes **errno**, **setjmp**, and **va_end**.

91. Since macro names are replaced whenever found, independent of scope and name space, macro names matching any of the reserved identifier names must not be defined if an associated header, if any, is included.

92. The macro **errno** need not be the identifier of an object. It might expand to a modifiable lvalue resulting from a function call (for example, ***errno()**).

93. Thus, a program that uses **errno** for error checking should set it to zero before a library function call, then inspect it before a subsequent library function call. Of course, a library function can save the value of **errno** on entry and then set it to zero, as long as the original value is restored if **errno**'s value is still zero just before the return.

Additional macro definitions, beginning with **E** and a digit or **E** and an uppercase letter,[94] may also be specified by the implementation.

4.1.4 Limits **<float.h>** and **<limits.h>**

5 The headers **<float.h>** and **<limits.h>** define several macros that expand to various limits and parameters.

The macros, their meanings, and the constraints (or restrictions) on their values are listed in 2.2.4.2.

4.1.5 Common Definitions **<stddef.h>**

The following types and macros are defined in the standard header **<stddef.h>**. Some are
10 also defined in other headers, as noted in their respective sections.

The types are

ptrdiff_t

which is the signed integral type of the result of subtracting two pointers;

size_t

15 which is the unsigned integral type of the result of the **sizeof** operator; and

wchar_t

which is an integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character shall have the code value zero and each member of the basic character set defined in 2.2.1 shall have a
20 code value equal to its value when used as the lone character in an integer character constant.

The macros are

NULL

which expands to an implementation-defined null pointer constant; and

offsetof(*type*, *member-designator***)**

25 which expands to an integral constant expression that has type **size_t**, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*). The *member-designator* shall be such that given

static *type* **t;**

then the expression **&(t**.*member-designator***)** evaluates to an address constant. (If the specified
30 member is a bit-field, the behavior is undefined.)

Forward references: localization (4.4).

---

94. See "future library directions" (4.13.1).

4.1.6 Use of Library Functions

Each of the following statements applies unless explicitly stated otherwise in the detailed descriptions that follow. If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer outside the address space of the program, or a
5   null pointer), the behavior is undefined. If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid. Any function declared in a header may be additionally implemented as a macro defined in the header, so a library function should not be declared explicitly if its header is
10   included. Any macro definition of a function can be suppressed locally by enclosing the name of the function in parentheses, because the name is then not followed by the left parenthesis that indicates expansion of a macro function name. For the same syntactic reason, it is permitted to take the address of a library function even if it is also defined as a macro.[95] The use of **#undef** to remove any macro definition will also ensure that an actual function is referred to. Any
15   invocation of a library function that is implemented as a macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments. Likewise, those function-like macros described in the following sections may be invoked in an expression anywhere a function with a compatible return type could be called.[96] All object-like macros listed as expanding to
20   integral constant expressions shall additionally be suitable for use in **#if** preprocessing directives.

Provided that a library function can be declared without reference to any type defined in a header, it is also permissible to declare the function, either explicitly or implicitly, and use it without including its associated header. If a function that accepts a variable number of arguments
25   is not declared (explicitly or by including its associated header), the behavior is undefined.

Examples

The function **atoi** may be used in any of several ways:

• by use of its associated header (possibly generating a macro expansion)

```
        #include <stdlib.h>
30      const char *str;
        /*...*/
        i = atoi(str);
```

---

95. This means that an implementation must provide an actual function for each library function, even if it also provides a macro for that function.

96. Because external identifiers and some macro names beginning with an underscore are reserved, implementations may provide special semantics for such names. For example, the identifier **_BUILTIN_abs** could be used to indicate generation of in-line code for the **abs** function. Thus, the appropriate header could specify

    #define abs(x) _BUILTIN_abs(x)

for a compiler whose code generator will accept it.

In this manner, a user desiring to guarantee that a given library function such as **abs** will be a genuine function may write

    #undef abs

whether the implementation's header provides a macro implementation of **abs** or a built-in implementation. The prototype for the function, which precedes and is hidden by any macro definition, is thereby revealed also.

- by use of its associated header (assuredly generating a true function reference)

```
          #include <stdlib.h>
          #undef atoi
          const char *str;
 5        /*...*/
          i = atoi(str);
    or
          #include <stdlib.h>
          const char *str;
10        /*...*/
          i = (atoi)(str);
```

- by explicit declaration

```
          extern int atoi(const char *);
          const char *str;
15        /*...*/
          i = atoi(str);
```

- by implicit declaration

```
          const char *str;
          /*...*/
20        i = atoi(str);
```

4.2 Diagnostics **<assert.h>**

The header **<assert.h>** defines the **assert** macro and refers to another macro,

NDEBUG

which is *not* defined by **<assert.h>**. If **NDEBUG** is defined as a macro name at the point in
5   the source file where **<assert.h>** is included, the **assert** macro is defined simply as

```
#define assert(ignore)  ((void)0)
```

The **assert** macro shall be implemented as a macro, not as an actual function. If the macro
definition is suppressed in order to access an actual function, the behavior is undefined.

4.2.1 Program Diagnostics

10 #### 4.2.1.1 The **assert** Macro

Synopsis

```
#include <assert.h>
void assert(int expression);
```

Description

15   The **assert** macro puts diagnostics into programs. When it is executed, if **expression** is
false (that is, compares equal to 0), the **assert** macro writes information about the particular
call that failed (including the text of the argument, the name of the source file, and the source
line number — the latter are respectively the values of the preprocessing macros **__FILE__** and
__LINE__) on the standard error file in an implementation-defined format.[97] It then calls the
20 **abort** function.

Returns

The **assert** macro returns no value.

Forward references: the **abort** function (4.10.4.1).

---

97. The message written might be of the form

Assertion failed: *expression*, file *xyz*, line *nnn*

4.3 Character Handling **<ctype.h>**

The header **<ctype.h>** declares several functions useful for testing and mapping characters.[98] In all cases the argument is an **int**, the value of which shall be representable as an **unsigned char** or shall equal the value of the macro **EOF**. If the argument has any other value, the behavior is undefined.

The behavior of these functions is affected by the current locale. Those functions that have implementation-defined aspects only when not in the "**C**" locale are noted below.

The term *printing character* refers to a member of an implementation-defined set of characters, each of which occupies one printing position on a display device; the term *control character* refers to a member of an implementation-defined set of characters that are not printing characters.[99]

Forward references: **EOF** (4.9.1), localization (4.4).

4.3.1 Character Testing Functions

The functions in this section return nonzero (true) if and only if the value of the argument **c** conforms to that in the description of the function.

4.3.1.1 The **isalnum** Function

Synopsis

```
#include <ctype.h>
int isalnum(int c);
```

Description

The **isalnum** function tests for any character for which **isalpha** or **isdigit** is true.

4.3.1.2 The **isalpha** Function

Synopsis

```
#include <ctype.h>
int isalpha(int c);
```

Description

The **isalpha** function tests for any character for which **isupper** or **islower** is true, or any character that is one of an implementation-defined set of characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true. In the "**C**" locale, **isalpha** returns true only for the characters for which **isupper** or **islower** is true.

4.3.1.3 The **iscntrl** Function

Synopsis

```
#include <ctype.h>
int iscntrl(int c);
```

---

98. See ''future library directions'' (4.13.2).

99. In an implementation that uses the seven-bit ASCII character set, the printing characters are those whose values lie from 0x20 (space) through 0x7E (tilde); the control characters are those whose values lie from 0 (NUL) through 0x1F (US), and the character 0x7F (DEL).

Description

>   The **iscntrl** function tests for any control character.

4.3.1.4 The **isdigit** Function

Synopsis

5
```
#include <ctype.h>
int isdigit(int c);
```

Description

>   The **isdigit** function tests for any decimal-digit character (as defined in 2.2.1).

4.3.1.5 The **isgraph** Function

10 **Synopsis**

```
#include <ctype.h>
int isgraph(int c);
```

Description

>   The **isgraph** function tests for any printing character except space (′ ′).

15 ### 4.3.1.6 The **islower** Function

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Description

20   The **islower** function tests for any character that is a lowercase letter or is one of an implementation-defined set of characters for which none of **iscntrl**, **isdigit**, **ispunct**, or **isspace** is true. In the "**C**" locale, **islower** returns true only for the characters defined as lowercase letters (as defined in 2.2.1).

4.3.1.7 The **isprint** Function

25 **Synopsis**

```
#include <ctype.h>
int isprint(int c);
```

Description

>   The **isprint** function tests for any printing character including space (′ ′).

30 ### 4.3.1.8 The **ispunct** Function

Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

Description

35   The **ispunct** function tests for any printing character that is neither space (′ ′) nor a character for which **isalnum** is true.

4.3.1.9 The `isspace` Function

Synopsis

```
#include <ctype.h>
int isspace(int c);
```

5 **Description**

The `isspace` function tests for any character that is a standard white-space character or is one of an implementation-defined set of characters for which `isalnum` is false. The standard white-space characters are the following: space (' '), form feed ('\f'), new-line ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v'). In the "C" locale, 10 `isspace` returns true only for the standard white-space characters.

4.3.1.10 The `isupper` Function

Synopsis

```
#include <ctype.h>
int isupper(int c);
```

15 **Description**

The `isupper` function tests for any character that is an uppercase letter or is one of an implementation-defined set of characters for which none of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true. In the "C" locale, `isupper` returns true only for the characters defined as uppercase letters (as defined in 2.2.1).

20 ### 4.3.1.11 The `isxdigit` Function

Synopsis

```
#include <ctype.h>
int isxdigit(int c);
```

Description

25 The `isxdigit` function tests for any hexadecimal-digit character (as defined in 3.1.3.2).

4.3.2 Character Case Mapping Functions

4.3.2.1 The `tolower` Function

Synopsis

```
#include <ctype.h>
30     int tolower(int c);
```

Description

The `tolower` function converts an uppercase letter to the corresponding lowercase letter.

Returns

If the argument is a character for which `isupper` is true and there is a corresponding 35 character for which `islower` is true, the `tolower` function returns the corresponding character; otherwise, the argument is returned unchanged.

4.3.2.2 The `toupper` Function

Synopsis

```
#include <ctype.h>
40     int toupper(int c);
```

Description

The **toupper** function converts a lowercase letter to the corresponding uppercase letter.

Returns

If the argument is a character for which **islower** is true and there is a corresponding
5   character for which **isupper** is true, the **toupper** function returns the corresponding character;
otherwise, the argument is returned unchanged.

4.4 Localization **<locale.h>**

The header **<locale.h>** declares two functions, one type, and defines several macros.

The type is

        struct lconv

5 which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges is explained in 4.4.2.1. In the **"C"** locale, the members shall have the values specified in the comments.

```
        char *decimal_point;        /* "." */
10      char *thousands_sep;        /* "" */
        char *grouping;             /* "" */
        char *int_curr_symbol;      /* "" */
        char *currency_symbol;      /* "" */
        char *mon_decimal_point;    /* "" */
15      char *mon_thousands_sep;    /* "" */
        char *mon_grouping;         /* "" */
        char *positive_sign;        /* "" */
        char *negative_sign;        /* "" */
        char int_frac_digits;       /* CHAR_MAX */
20      char frac_digits;           /* CHAR_MAX */
        char p_cs_precedes;         /* CHAR_MAX */
        char p_sep_by_space;        /* CHAR_MAX */
        char n_cs_precedes;         /* CHAR_MAX */
        char n_sep_by_space;        /* CHAR_MAX */
25      char p_sign_posn;           /* CHAR_MAX */
        char n_sign_posn;           /* CHAR_MAX */
```

The macros defined are **NULL** (described in 4.1.5); and

```
        LC_ALL
        LC_COLLATE
30      LC_CTYPE
        LC_MONETARY
        LC_NUMERIC
        LC_TIME
```

which expand to integral constant expressions with distinct values, suitable for use as the first
35 argument to the **setlocale** function. Additional macro definitions, beginning with the characters **LC_** and an uppercase letter,[100] may also be specified by the implementation.

---

100. See ''future library directions'' (4.13.3).

4.4.1 Locale Control

4.4.1.1 The `setlocale` Function

Synopsis

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Description

The **setlocale** function selects the appropriate portion of the program's locale as specified by the **category** and **locale** arguments. The **setlocale** function may be used to change or query the program's entire current locale or portions thereof. The value **LC_ALL** for **category** names the program's entire locale; the other values for **category** name only a portion of the program's locale. **LC_COLLATE** affects the behavior of the **strcoll** and **strxfrm** functions. **LC_CTYPE** affects the behavior of the character handling functions[101] and the multibyte functions. **LC_MONETARY** affects the monetary formatting information returned by the **localeconv** function. **LC_NUMERIC** affects the decimal-point character for the formatted input/output functions and the string conversion functions, as well as the nonmonetary formatting information returned by the **localeconv** function. **LC_TIME** affects the behavior of the **strftime** function.

A value of `"C"` for **locale** specifies the minimal environment for C translation; a value of `""` for **locale** specifies the implementation-defined native environment. Other implementation-defined strings may be passed as the second argument to **setlocale**.

At program startup, the equivalent of

```
setlocale(LC_ALL, "C");
```

is executed.

The implementation shall behave as if no library function calls the **setlocale** function.

Returns

If a pointer to a string is given for **locale** and the selection can be honored, the **setlocale** function returns a pointer to the string associated with the specified **category** for the new locale. If the selection cannot be honored, the **setlocale** function returns a null pointer and the program's locale is not changed.

A null pointer for **locale** causes the **setlocale** function to return a pointer to the string associated with the **category** for the program's current locale; the program's locale is not changed.[102]

The pointer to string returned by the **setlocale** function is such that a subsequent call with that string value and its associated category will restore that part of the program's locale. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **setlocale** function.

Forward references: formatted input/output functions (4.9.6), the multibyte character functions (4.10.7), the multibyte string functions (4.10.8), string conversion functions (4.10.1), the **strcoll** function (4.11.4.3), the **strftime** function (4.12.3.5), the **strxfrm** function (4.11.4.5).

---

101. The only functions in 4.3 whose behavior is not affected by the current locale are **isdigit** and **isxdigit**.

102. The implementation must arrange to encode in a string the various categories due to a heterogeneous locale when **category** has the value **LC_ALL**.

4.4.2 Numeric Formatting Convention Inquiry

4.4.2.1 The `localeconv` Function

Synopsis

```
#include <locale.h>
struct lconv *localeconv(void);
```
5

Description

The **localeconv** function sets the components of an object with type **struct lconv** with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

10 The members of the structure with type **char *** are pointers to strings, any of which (except **decimal_point**) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type **char** are nonnegative numbers, any of which can be **CHAR_MAX** to indicate that the value is not available in the current locale. The members include the following:

15 **char *decimal_point**
        The decimal-point character used to format nonmonetary quantities.

char *thousands_sep
        The character used to separate groups of digits before the decimal-point character in formatted nonmonetary quantities.

20 **char *grouping**
        A string whose elements indicate the size of each group of digits in formatted nonmonetary quantities.

char *int_curr_symbol
        The international currency symbol applicable to the current locale. The first three
25   characters contain the alphabetic international currency symbol in accordance with those specified in ISO 4217:1987. The fourth character (immediately preceding the null character) is the character used to separate the international currency symbol from the monetary quantity.

char *currency_symbol
30   The local currency symbol applicable to the current locale.

char *mon_decimal_point
        The decimal-point used to format monetary quantities.

char *mon_thousands_sep
        The separator for groups of digits before the decimal-point in formatted monetary
35   quantities.

char *mon_grouping
        A string whose elements indicate the size of each group of digits in formatted monetary quantities.

char *positive_sign
40   The string used to indicate a nonnegative-valued formatted monetary quantity.

char *negative_sign
        The string used to indicate a negative-valued formatted monetary quantity.

char int_frac_digits
        The number of fractional digits (those after the decimal-point) to be displayed in a
45   internationally formatted monetary quantity.

char frac_digits

> The number of fractional digits (those after the decimal-point) to be displayed in a formatted monetary quantity.

char p_cs_precedes

5

> Set to 1 or 0 if the **currency_symbol** respectively precedes or succeeds the value for a nonnegative formatted monetary quantity.

char p_sep_by_space

> Set to 1 or 0 if the **currency_symbol** respectively is or is not separated by a space from the value for a nonnegative formatted monetary quantity.

10 **char n_cs_precedes**

> Set to 1 or 0 if the **currency_symbol** respectively precedes or succeeds the value for a negative formatted monetary quantity.

char n_sep_by_space

15

> Set to 1 or 0 if the **currency_symbol** respectively is or is not separated by a space from the value for a negative formatted monetary quantity.

char p_sign_posn

> Set to a value indicating the positioning of the **positive_sign** for a nonnegative formatted monetary quantity.

char n_sign_posn

20

> Set to a value indicating the positioning of the **negative_sign** for a negative formatted monetary quantity.

The elements of **grouping** and **mon_grouping** are interpreted according to the following:

CHAR_MAX   No further grouping is to be performed.

0   The previous element is to be repeatedly used for the remainder of the digits.

25 *other*   The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The value of **p_sign_posn** and **n_sign_posn** is interpreted according to the following:

0   Parentheses surround the quantity and **currency_symbol**.

30 **1**   The sign string precedes the quantity and **currency_symbol**.

2   The sign string succeeds the quantity and **currency_symbol**.

3   The sign string immediately precedes the **currency_symbol**.

4   The sign string immediately succeeds the **currency_symbol**.

The implementation shall behave as if no library function calls the **localeconv** function.

35 **Returns**

The **localeconv** function returns a pointer to the filled-in object. The structure pointed to by the return value shall not be modified by the program, but may be overwritten by a subsequent call to the **localeconv** function. In addition, calls to the **setlocale** function with categories **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** may overwrite the contents of the

40 structure.

Examples

The following table illustrates the rules which may well be used by four countries to format monetary quantities.

Country	Positive format	Negative format	International format
Italy	`L.1.234`	`-L.1.234`	`ITL.1.234`
Netherlands	`F 1.234,56`	`F -1.234,56`	`NLG 1.234,56`
Norway	`kr1.234,56`	`kr1.234,56-`	`NOK 1.234,56`
Switzerland	`SFrs.1,234.56`	`SFrs.1,234.56C`	`CHF 1,234.56`

For these four countries, the respective values for the monetary members of the structure returned by **localeconv** are:

	Italy	Netherlands	Norway	Switzerland
`int_curr_symbol`	`"ITL."`	`"NLG "`	`"NOK "`	`"CHF "`
`currency_symbol`	`"L."`	`"F"`	`"kr"`	`"SFrs."`
`mon_decimal_point`	`""`	`","`	`","`	`"."`
`mon_thousands_sep`	`"."`	`"."`	`"."`	`","`
`mon_grouping`	`"\3"`	`"\3"`	`"\3"`	`"\3"`
`positive_sign`	`""`	`""`	`""`	`""`
`negative_sign`	`"-"`	`"-"`	`"-"`	`"C"`
`int_frac_digits`	0	2	2	2
`frac_digits`	0	2	2	2
`p_cs_precedes`	1	1	1	1
`p_sep_by_space`	0	1	0	0
`n_cs_precedes`	1	1	1	1
`n_sep_by_space`	0	1	0	0
`p_sign_posn`	1	1	1	1
`n_sign_posn`	1	4	2	2

4.5 Mathematics \<math.h\>

The header **\<math.h\>** declares several mathematical functions and defines one macro. The functions take **double** arguments and return **double** values.[103] Integer arithmetic functions and conversion functions are discussed later.

5   The macro defined is

      HUGE_VAL

which expands to a positive **double** expression, not necessarily representable as a **float**.[104]

Forward references: integer arithmetic functions (4.10.6), the **atof** function (4.10.1.1), the **strtod** function (4.10.1.4).

10   ### 4.5.1 Treatment of Error Conditions

The behavior of each of these functions is defined for all representable values of its input arguments. Each function shall execute as if it were a single operation, without generating any externally visible exceptions.

For all functions, a *domain error* occurs if an input argument is outside the domain over
15   which the mathematical function is defined. The description of each function lists any required domain errors; an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of the function.[105] On a domain error, the function returns an implementation-defined value; the value of the macro **EDOM** is stored in **errno**.

20   Similarly, a *range error* occurs if the result of the function cannot be represented as a **double** value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro **HUGE_VAL**, with the same sign (except for the **tan** function) as the correct value of the function; the value of the macro **ERANGE** is stored in **errno**. If the result underflows (the
25   magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero; whether the integer expression **errno** acquires the value of the macro **ERANGE** is implementation-defined.

4.5.2 Trigonometric Functions

4.5.2.1 The acos Function

30   **Synopsis**

```
#include <math.h>
double acos(double x);
```

Description

The **acos** function computes the principal value of the arc cosine of **x**. A domain error
35   occurs for arguments not in the range [−1, +1].

---

103. See "future library directions" (4.13.4).

104. **HUGE_VAL** can be positive infinity in an implementation that supports infinities.

105. In an implementation that supports infinities, this allows infinity as an argument to be a domain error if the mathematical domain of the function does not include infinity.

Returns

The **acos** function returns the arc cosine in the range [0, π] radians.

4.5.2.2 The **asin** Function

Synopsis

5
```
#include <math.h>
double asin(double x);
```

Description

The **asin** function computes the principal value of the arc sine of **x**. A domain error occurs for arguments not in the range [−1, +1].

10 **Returns**

The **asin** function returns the arc sine in the range [−π/2, +π/2] radians.

4.5.2.3 The **atan** Function

Synopsis

```
#include <math.h>
```
15
```
double atan(double x);
```

Description

The **atan** function computes the principal value of the arc tangent of **x**.

Returns

The **atan** function returns the arc tangent in the range [−π/2, +π/2] radians.

20 ### 4.5.2.4 The **atan2** Function

Synopsis

```
#include <math.h>
double atan2(double y, double x);
```

Description

25   The **atan2** function computes the principal value of the arc tangent of **y/x**, using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.

Returns

The **atan2** function returns the arc tangent of **y/x**, in the range [−π, +π] radians.

30 ### 4.5.2.5 The **cos** Function

Synopsis

```
#include <math.h>
double cos(double x);
```

Description

35   The **cos** function computes the cosine of **x** (measured in radians).

Returns

The **cos** function returns the cosine value.

4.5.2.6 The `sin` Function

Synopsis

```
#include <math.h>
double sin(double x);
```

5   Description

The `sin` function computes the sine of **x** (measured in radians).

Returns

The `sin` function returns the sine value.

4.5.2.7 The `tan` Function

10   Synopsis

```
#include <math.h>
double tan(double x);
```

Description

The `tan` function returns the tangent of **x** (measured in radians).

15   Returns

The `tan` function returns the tangent value.

4.5.3 Hyperbolic Functions

4.5.3.1 The `cosh` Function

Synopsis

20
```
#include <math.h>
double cosh(double x);
```

Description

The `cosh` function computes the hyperbolic cosine of **x**. A range error occurs if the magnitude of **x** is too large.

25   Returns

The `cosh` function returns the hyperbolic cosine value.

4.5.3.2 The `sinh` Function

Synopsis

```
#include <math.h>
30   double sinh(double x);
```

Description

The `sinh` function computes the hyperbolic sine of **x**. A range error occurs if the magnitude of **x** is too large.

Returns

35   The `sinh` function returns the hyperbolic sine value.

4.5.3.3 The `tanh` Function

Synopsis

```
#include <math.h>
double tanh(double x);
```

5   **Description**

The **tanh** function computes the hyperbolic tangent of **x**.

Returns

The **tanh** function returns the hyperbolic tangent value.

4.5.4 Exponential and Logarithmic Functions

10   ### 4.5.4.1 The `exp` Function

Synopsis

```
#include <math.h>
double exp(double x);
```

Description

15   The **exp** function computes the exponential function of **x**. A range error occurs if the magnitude of **x** is too large.

Returns

The **exp** function returns the exponential value.

4.5.4.2 The `frexp` Function

20   **Synopsis**

```
#include <math.h>
double frexp(double value, int *exp);
```

Description

The **frexp** function breaks a floating-point number into a normalized fraction and an integral 25   power of 2. It stores the integer in the **int** object pointed to by **exp**.

Returns

The **frexp** function returns the value **x**, such that **x** is a **double** with magnitude in the interval [1/2, 1) or zero, and **value** equals **x** times 2 raised to the power ***exp**. If **value** is zero, both parts of the result are zero.

30   ### 4.5.4.3 The `ldexp` Function

Synopsis

```
#include <math.h>
double ldexp(double x, int exp);
```

Description

35   The **ldexp** function multiplies a floating-point number by an integral power of 2. A range error may occur.

Returns

The **ldexp** function returns the value of **x** times 2 raised to the power **exp**.

4.5.4.4 The `log` Function

Synopsis

```
#include <math.h>
double log(double x);
```

5   Description

The **log** function computes the natural logarithm of **x**. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

Returns

The **log** function returns the natural logarithm.

10 ### 4.5.4.5 The `log10` Function

Synopsis

```
#include <math.h>
double log10(double x);
```

Description

15   The **log10** function computes the base-ten logarithm of **x**. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

Returns

The **log10** function returns the base-ten logarithm.

4.5.4.6 The `modf` Function

20 Synopsis

```
#include <math.h>
double modf(double value, double *iptr);
```

Description

The **modf** function breaks the argument **value** into integral and fractional parts, each of
25 which has the same sign as the argument. It stores the integral part as a **double** in the object pointed to by **iptr**.

Returns

The **modf** function returns the signed fractional part of **value**.

4.5.5 Power Functions

30 ### 4.5.5.1 The `pow` Function

Synopsis

```
#include <math.h>
double pow(double x, double y);
```

Description

35   The **pow** function computes **x** raised to the power **y**. A domain error occurs if **x** is negative and **y** is not an integral value. A domain error occurs if the result cannot be represented when **x** is zero and **y** is less than or equal to zero. A range error may occur.

Returns

The **pow** function returns the value of **x** raised to the power **y**.

4.5.5.2 The sqrt Function

Synopsis

```
#include <math.h>
double sqrt(double x);
```

5 **Description**

The **sqrt** function computes the nonnegative square root of **x**. A domain error occurs if the argument is negative.

Returns

The **sqrt** function returns the value of the square root.

10 ## 4.5.6 Nearest Integer, Absolute Value, and Remainder Functions

4.5.6.1 The ceil Function

Synopsis

```
#include <math.h>
double ceil(double x);
```

15 **Description**

The **ceil** function computes the smallest integral value not less than **x**.

Returns

The **ceil** function returns the smallest integral value not less than **x**, expressed as a double.

4.5.6.2 The fabs Function

20 **Synopsis**

```
#include <math.h>
double fabs(double x);
```

Description

The **fabs** function computes the absolute value of a floating-point number **x**.

25 **Returns**

The **fabs** function returns the absolute value of **x**.

4.5.6.3 The floor Function

Synopsis

```
#include <math.h>
30      double floor(double x);
```

Description

The **floor** function computes the largest integral value not greater than **x**.

Returns

The **floor** function returns the largest integral value not greater than **x**, expressed as a
35 double.

4.5.6.4 The **fmod** Function

Synopsis

```
#include <math.h>
double fmod(double x, double y);
```

5 **Description**

The **fmod** function computes the floating-point remainder of **x/y**.

Returns

The **fmod** function returns the value $x - i * y$, for some integer i such that, if **y** is nonzero, the result has the same sign as **x** and magnitude less than the magnitude of **y**. If **y** is zero,
10 whether a domain error occurs or the **fmod** function returns zero is implementation-defined.

4.6 Nonlocal Jumps `<setjmp.h>`

The header `<setjmp.h>` defines the macro `setjmp`, and declares one function and one type, for bypassing the normal function call and return discipline.[106]

The type declared is

5   `jmp_buf`

which is an array type suitable for holding the information needed to restore a calling environment.

It is unspecified whether `setjmp` is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an
10   external identifier with the name `setjmp`, the behavior is undefined.

4.6.1 Save Calling Environment

4.6.1.1 The `setjmp` Macro

Synopsis

```
         #include <setjmp.h>
15       int setjmp(jmp_buf env);
```

Description

The `setjmp` macro saves its calling environment in its `jmp_buf` argument for later use by the `longjmp` function.

Returns

20   If the return is from a direct invocation, the `setjmp` macro returns the value zero. If the return is from a call to the `longjmp` function, the `setjmp` macro returns a nonzero value.

Environmental Constraint

An invocation of the `setjmp` macro shall appear only in one of the following contexts:

- the entire controlling expression of a selection or iteration statement;

25   - one operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement;

- the operand of a unary `!` operator with the resulting expression being the entire controlling expression of a selection or iteration statement; or

30   - the entire expression of an expression statement (possibly cast to **void**).

---

106. These functions are useful for dealing with unusual conditions encountered in a low-level function of a program.

4.6.2 Restore Calling Environment

4.6.2.1 The `longjmp` Function

Synopsis

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Description

The **longjmp** function restores the environment saved by the most recent invocation of the **setjmp** macro in the same invocation of the program, with the corresponding **jmp_buf** argument. If there has been no such invocation, or if the function containing the invocation of the **setjmp** macro has terminated execution[107] in the interim, the behavior is undefined.

All accessible objects have values as of the time **longjmp** was called, except that the values of objects of automatic storage duration that are local to the function containing the invocation of the corresponding **setjmp** macro that do not have volatile-qualified type and have been changed between the **setjmp** invocation and **longjmp** call are indeterminate.

As it bypasses the usual function call and return mechanisms, the **longjmp** function shall execute correctly in contexts of interrupts, signals and any of their associated functions. However, if the **longjmp** function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

Returns

After **longjmp** is completed, program execution continues as if the corresponding invocation of the **setjmp** macro had just returned the value specified by **val**. The **longjmp** function cannot cause the **setjmp** macro to return the value 0; if **val** is 0, the **setjmp** macro returns the value 1.

---

107. For example, by executing a **return** statement or because another **longjmp** call has caused a transfer to a **setjmp** invocation in a function earlier in the set of nested calls.

4.7 Signal Handling `<signal.h>`

The header `<signal.h>` declares a type and two functions and defines several macros, for handling various *signals* (conditions that may be reported during program execution).

The type defined is

5   `sig_atomic_t`

which is the integral type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

The macros defined are

```
       SIG_DFL
10     SIG_ERR
       SIG_IGN
```

which expand to constant expressions with distinct values that have type compatible with the second argument to and the return value of the **signal** function, and whose value compares unequal to the address of any declarable function; and the following, each of which expands to a
15 positive integral constant expression that is the signal number corresponding to the specified condition:

SIGABRT abnormal termination, such as is initiated by the **abort** function

SIGFPE   an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow

20 **SIGILL**   detection of an invalid function image, such as an illegal instruction

SIGINT   receipt of an interactive attention signal

SIGSEGV an invalid access to storage

SIGTERM a termination request sent to the program

An implementation need not generate any of these signals, except as a result of explicit calls
25 to the **raise** function. Additional signals and pointers to undeclarable functions, with macro definitions beginning, respectively, with the letters **SIG** and an uppercase letter or with **SIG_** and an uppercase letter,[108] may also be specified by the implementation. The complete set of signals, their semantics, and their default handling is implementation-defined; all signal numbers shall be positive.

30 ## 4.7.1 Specify Signal Handling

4.7.1.1 The `signal` Function

Synopsis

```
       #include <signal.h>
       void (*signal(int sig, void (*func)(int)))(int);
```

35 **Description**

The **signal** function chooses one of three ways in which receipt of the signal number **sig** is to be subsequently handled. If the value of **func** is **SIG_DFL**, default handling for that signal will occur. If the value of **func** is **SIG_IGN**, the signal will be ignored. Otherwise,

---

108. See "future library directions" (4.13.5). The names of the signal numbers reflect the following terms (respectively): abort, floating-point exception, illegal instruction, interrupt, segmentation violation, and termination.

func shall point to a function to be called when that signal occurs. Such a function is called a *signal handler*.

When a signal occurs, if **func** points to a function, first the equivalent of **signal(sig, SIG_DFL);** is executed or an implementation-defined blocking of the signal is performed. (If
5 the value of **sig** is **SIGILL**, whether the reset to **SIG_DFL** occurs is implementation-defined.) Next the equivalent of **(*func)(sig);** is executed. The function **func** may terminate by executing a **return** statement or by calling the **abort**, **exit**, or **longjmp** function. If **func** executes a **return** statement and the value of **sig** was **SIGFPE** or any other implementation-defined value corresponding to a computational exception, the behavior is undefined. Otherwise,
10 the program will resume execution at the point it was interrupted.

If the signal occurs other than as the result of calling the **abort** or **raise** function, the behavior is undefined if the signal handler calls any function in the standard library other than the **signal** function itself (with a first argument of the signal number corresponding to the signal that caused the invocation of the handler) or refers to any object with static storage duration other
15 than by assigning a value to a static storage duration variable of type **volatile sig_atomic_t**. Furthermore, if such a call to the **signal** function results in a **SIG_ERR** return, the value of **errno** is indeterminate.[109]

At program startup, the equivalent of

        signal(sig, SIG_IGN);

20 may be executed for some signals selected in an implementation-defined manner; the equivalent of

        signal(sig, SIG_DFL);

is executed for all other signals defined by the implementation.

The implementation shall behave as if no library function calls the **signal** function.

25 **Returns**

If the request can be honored, the **signal** function returns the value of **func** for the most recent call to **signal** for the specified signal **sig**. Otherwise, a value of **SIG_ERR** is returned and a positive value is stored in **errno**.

Forward references: the **abort** function (4.10.4.1), the **exit** function (4.10.4.3).

30 **4.7.2 Send Signal**

4.7.2.1 The raise Function

Synopsis

        #include <signal.h>
        int raise(int sig);

35 **Description**

The **raise** function sends the signal **sig** to the executing program.

Returns

The **raise** function returns zero if successful, nonzero if unsuccessful.

---

109. If any signal is generated by an asynchronous signal handler, the behavior is undefined.

4.8 Variable Arguments **\<stdarg.h\>**

The header **\<stdarg.h\>** declares a type and defines three macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

5   A function may be called with a variable number of arguments of varying types. As described in 3.7.1, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated *parmN* in this description.

The type declared is

        va_list

10   which is a type suitable for holding information needed by the macros **va_start**, **va_arg**, and **va_end**. If access to the varying arguments is desired, the called function shall declare an object (referred to as **ap** in this section) having type **va_list**. The object **ap** may be passed as an argument to another function; if that function invokes the **va_arg** macro with parameter **ap**, the value of **ap** in the calling function is indeterminate and shall be passed to the **va_end** macro 15   prior to any further reference to **ap**.

4.8.1 Variable Argument List Access Macros

The **va_start** and **va_arg** macros described in this section shall be implemented as macros, not as actual functions. It is unspecified whether **va_end** is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual 20   function, or a program defines an external identifier with the name **va_end**, the behavior is undefined. The **va_start** and **va_end** macros shall be invoked in the function accepting a varying number of arguments, if access to the varying arguments is desired.

4.8.1.1 The va_start Macro

Synopsis

25   ```
        #include <stdarg.h>
        void va_start(va_list ap, parmN);
        ```

**Description**

The **va_start** macro shall be invoked before any access to the unnamed arguments.

The **va_start** macro initializes **ap** for subsequent use by **va_arg** and **va_end**.

30      The parameter *parmN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...). If the parameter *parmN* is declared with the **register** storage class, 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, the behavior is undefined.

35   **Returns**

The **va_start** macro returns no value.

#### 4.8.1.2  The va_arg Macro

**Synopsis**

        ```
        #include <stdarg.h>
40      type va_arg(va_list ap, type);
        ```

Description

The **va_arg** macro expands to an expression that has the type and value of the next argument in the call. The parameter **ap** shall be the same as the **va_list ap** initialized by **va_start**. Each invocation of **va_arg** modifies **ap** so that the values of successive arguments

are returned in turn. The parameter *type* is a type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by postfixing a * to *type*. If there is no actual next argument, or if *type* is not compatible with the type of the actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

5 **Returns**

The first invocation of the **va_arg** macro after that of the **va_start** macro returns the value of the argument after that specified by *parmN*. Successive invocations return the values of the remaining arguments in succession.

4.8.1.3 The va_end Macro

10 **Synopsis**

```
#include <stdarg.h>
void va_end(va_list ap);
```

Description

The **va_end** macro facilitates a normal return from the function whose variable argument list
15 was referred to by the expansion of **va_start** that initialized the **va_list ap**. The **va_end** macro may modify **ap** so that it is no longer usable (without an intervening invocation of **va_start**). If there is no corresponding invocation of the **va_start** macro, or if the **va_end** macro is not invoked before the return, the behavior is undefined.

Returns

20 The **va_end** macro returns no value.

Example

The function **f1** gathers into an array a list of arguments that are pointers to strings (but not more than **MAXARGS** arguments), then passes the array as a single argument to function **f2**. The number of pointers is specified by the first argument to **f1**.

25
```
#include <stdarg.h>
#define MAXARGS   31

void f1(int n_ptrs, ...)
{
        va_list ap;
        char *array[MAXARGS];
        int ptr_no = 0;

        if (n_ptrs > MAXARGS)
                n_ptrs = MAXARGS;
        va_start(ap, n_ptrs);
        while (ptr_no < n_ptrs)
                array[ptr_no++] = va_arg(ap, char *);
        va_end(ap);
        f2(n_ptrs, array);
}
```

40 Each call to **f1** shall have visible the definition of the function or a declaration such as

```
void f1(int, ...);
```

4.9 Input/Output **<stdio.h>**

4.9.1 Introduction

The header **<stdio.h>** declares three types, several macros, and many functions for performing input and output.

5   The types declared are **size_t** (described in 4.1.5);

> **FILE**

which is an object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records
10   whether the end of the file has been reached; and

> **fpos_t**

which is an object type capable of recording all the information needed to specify uniquely every position within a file.

The macros are **NULL** (described in 4.1.5);

15   **_IOFBF**
          _IOLBF
          _IONBF

which expand to integral constant expressions with distinct values, suitable for use as the third argument to the **setvbuf** function;

20   **BUFSIZ**

which expands to an integral constant expression, which is the size of the buffer used by the **setbuf** function;

> **EOF**

which expands to a negative integral constant expression that is returned by several functions to
25   indicate *end-of-file*, that is, no more input from a stream;

> **FOPEN_MAX**

which expands to an integral constant expression that is the minimum number of files that the implementation guarantees can be open simultaneously;

> **FILENAME_MAX**

30   which expands to an integral constant expression that is the size needed for an array of **char** large enough to hold the longest file name string that the implementation guarantees can be opened;[110]

> **L_tmpnam**

which expands to an integral constant expression that is the size needed for an array of **char**
35   large enough to hold a temporary file name string generated by the **tmpnam** function;

---

110. If the implementation imposes no practical limit on the length of file name strings, the value of **FILENAME_MAX** should instead be the recommended size of an array intended to hold a file name string. Of course, file name string contents are subject to other system-specific constraints; therefore *all* possible strings of length **FILENAME_MAX** cannot be expected to be opened successfully.

      SEEK_CUR
      SEEK_END
      SEEK_SET

which expand to integral constant expressions with distinct values, suitable for use as the third argument to the **fseek** function;

      TMP_MAX

which expands to an integral constant expression that is the minimum number of unique file names that shall be generated by the **tmpnam** function;

      stderr
      stdin
      stdout

which are expressions of type "pointer to **FILE**" that point to the **FILE** objects associated, respectively, with the standard error, input, and output streams.

Forward references: files (4.9.3), the **fseek** function (4.9.9.2), streams (4.9.2), the **tmpnam** function (4.9.4.4).

4.9.2 Streams

Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data *streams*, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported, for *text streams* and for *binary streams*.[111]

A text stream is an ordered sequence of characters composed into *lines*, each line consisting of zero or more characters plus a terminating new-line character. Whether the last line requires a terminating new-line character is implementation-defined. Characters may have to be added, altered, or deleted on input and output to conform to differing conventions for representing text in the host environment. Thus, there need not be a one-to-one correspondence between the characters in a stream and those in the external representation. Data read in from a text stream will necessarily compare equal to the data that were earlier written out to that stream only if: the data consist only of printable characters and the control characters horizontal tab and new-line; no new-line character is immediately preceded by space characters; and the last character is a new-line character. Whether space characters that are written out immediately before a new-line character appear when read in is implementation-defined.

A binary stream is an ordered sequence of characters that can transparently record internal data. Data read in from a binary stream shall compare equal to the data that were earlier written out to that stream, under the same implementation. Such a stream may, however, have an implementation-defined number of null characters appended to the end of the stream.

Environmental Limits

An implementation shall support text files with lines containing at least 254 characters, including the terminating new-line character. The value of the macro **BUFSIZ** shall be at least 256.

---

111. An implementation need not distinguish between text streams and binary streams. In such an implementation, there need be no new-line characters in a text stream nor any limit to the length of a line.

4.9.3 Files

A stream is associated with an external file (which may be a physical device) by *opening* a file, which may involve *creating* a new file. Creating an existing file causes its former contents to be discarded, if necessary. If a file can support positioning requests (such as a disk file, as 5 opposed to a terminal), then a *file position indicator*[112] associated with the stream is positioned at the start (character number zero) of the file, unless the file is opened with append mode in which case it is implementation-defined whether the file position indicator is initially positioned at the beginning or the end of the file. The file position indicator is maintained by subsequent reads, writes, and positioning requests, to facilitate an orderly progression through the file. All input 10 takes place as if characters were read by successive calls to the **fgetc** function; all output takes place as if characters were written by successive calls to the **fputc** function.

Binary files are not truncated, except as defined in 4.9.5.3. Whether a write on a text stream causes the associated file to be truncated beyond that point is implementation-defined.

When a stream is *unbuffered*, characters are intended to appear from the source or at the 15 destination as soon as possible. Otherwise characters may be accumulated and transmitted to or from the host environment as a block. When a stream is *fully buffered*, characters are intended to be transmitted to or from the host environment as a block when a buffer is filled. When a stream is *line buffered*, characters are intended to be transmitted to or from the host environment as a block when a new-line character is encountered. Furthermore, characters are intended to be 20 transmitted as a block to the host environment when a buffer is filled, when input is requested on an unbuffered stream, or when input is requested on a line buffered stream that requires the transmission of characters from the host environment. Support for these characteristics is implementation-defined, and may be affected via the **setbuf** and **setvbuf** functions.

A file may be disassociated from a controlling stream by *closing* the file. Output streams are 25 flushed (any unwritten buffer contents are transmitted to the host environment) before the stream is disassociated from the file. The value of a pointer to a **FILE** object is indeterminate after the associated file is closed (including the standard text streams). Whether a file of zero length (on which no characters have been written by an output stream) actually exists is implementation-defined.

30 The file may be subsequently reopened, by the same or another program execution, and its contents reclaimed or modified (if it can be repositioned at its start). If the **main** function returns to its original caller, or if the **exit** function is called, all open files are closed (hence all output streams are flushed) before program termination. Other paths to program termination, such as calling the **abort** function, need not close all files properly.

35 The address of the **FILE** object used to control a stream may be significant; a copy of a **FILE** object may not necessarily serve in place of the original.

At program startup, three text streams are predefined and need not be opened explicitly — *standard input* (for reading conventional input), *standard output* (for writing conventional output), and *standard error* (for writing diagnostic output). When opened, the standard error 40 stream is not fully buffered; the standard input and standard output streams are fully buffered if and only if the stream can be determined not to refer to an interactive device.

Functions that open additional (nontemporary) files require a *file name*, which is a string. The rules for composing valid file names are implementation-defined. Whether the same file can be simultaneously open multiple times is also implementation-defined.

---

112. This is described in the Base Document as a *file pointer*. That term is not used in this standard to avoid confusion with a pointer to an object that has type **FILE**.

Environmental Limits

The value of **FOPEN_MAX** shall be at least eight, including the three standard text streams.

Forward references: the **exit** function (4.10.4.3), the **fgetc** function (4.9.7.1), the **fopen** function (4.9.5.3), the **fputc** function (4.9.7.3), the **setbuf** function (4.9.5.5), the **setvbuf**
5   function (4.9.5.6).

4.9.4 Operations on Files

4.9.4.1 The **remove** Function

Synopsis

```
        #include <stdio.h>
10      int remove(const char *filename);
```

Description

The **remove** function causes the file whose name is the string pointed to by **filename** to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the **remove** function is
15   implementation-defined.

Returns

The **remove** function returns zero if the operation succeeds, nonzero if it fails.

4.9.4.2 The **rename** Function

Synopsis

```
20      #include <stdio.h>
        int rename(const char *old, const char *new);
```

Description

The **rename** function causes the file whose name is the string pointed to by **old** to be henceforth known by the name given by the string pointed to by **new**. The file named **old** is no
25   longer accessible by that name. If a file named by the string pointed to by **new** exists prior to the call to the **rename** function, the behavior is implementation-defined.

Returns

The **rename** function returns zero if the operation succeeds, nonzero if it fails,[113] in which case if the file existed previously it is still known by its original name.

30   ### 4.9.4.3 The **tmpfile** Function

Synopsis

```
        #include <stdio.h>
        FILE *tmpfile(void);
```

Description

35   The **tmpfile** function creates a temporary binary file that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with "**wb+**" mode.

---

113. Among the reasons the implementation may cause the **rename** function to fail are that the file is open or that it is necessary to copy its contents to effectuate its renaming.

Returns

The **tmpfile** function returns a pointer to the stream of the file that it created. If the file cannot be created, the **tmpfile** function returns a null pointer.

Forward references: the **fopen** function (4.9.5.3).

5 **4.9.4.4 The tmpnam Function**

Synopsis

```
#include <stdio.h>
char *tmpnam(char *s);
```

Description

10   The **tmpnam** function generates a string that is a valid file name and that is not the same as the name of an existing file.[114]

The **tmpnam** function generates a different string each time it is called, up to **TMP_MAX** times. If it is called more than **TMP_MAX** times, the behavior is implementation-defined.

The implementation shall behave as if no library function calls the **tmpnam** function.

15 **Returns**

If the argument is a null pointer, the **tmpnam** function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to the **tmpnam** function may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least **L_tmpnam char**s; the **tmpnam** function writes its result in that array and returns the
20 argument as its value.

Environmental Limits

The value of the macro **TMP_MAX** shall be at least 25.

4.9.5 File Access Functions

4.9.5.1 The fclose Function

25 **Synopsis**

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description

The **fclose** function causes the stream pointed to by **stream** to be flushed and the
30 associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.

---

114. Files created using strings generated by the **tmpnam** function are temporary only in the sense that their names should not collide with those generated by conventional naming rules for the implementation. It is still necessary to use the **remove** function to remove such files when their use is ended, and before program termination.

Returns

The **fclose** function returns zero if the stream was successfully closed, or **EOF** if any errors were detected.

4.9.5.2 The **fflush** Function

5 Synopsis

```
#include <stdio.h>
int fflush(FILE *stream);
```

Description

If **stream** points to an output stream or an update stream in which the most recent operation
10 was not input, the **fflush** function causes any unwritten data for that stream to be delivered to
the host environment to be written to the file; otherwise, the behavior is undefined.

If **stream** is a null pointer, the **fflush** function performs this flushing action on all
streams for which the behavior is defined above.

Returns

15   The **fflush** function returns **EOF** if a write error occurs, otherwise zero.

Forward references: the **fopen** function (4.9.5.3), the **ungetc** function (4.9.7.11).

4.9.5.3 The **fopen** Function

Synopsis

```
#include <stdio.h>
20      FILE *fopen(const char *filename, const char *mode);
```

Description

The **fopen** function opens the file whose name is the string pointed to by **filename**, and
associates a stream with it.

The argument **mode** points to a string beginning with one of the following sequences:[115]

25 **r**   open text file for reading
    w   truncate to zero length or create text file for writing
    a   append; open or create text file for writing at end-of-file
    rb   open binary file for reading
    wb   truncate to zero length or create binary file for writing
30 **ab**   append; open or create binary file for writing at end-of-file
    r+   open text file for update (reading and writing)
    w+   truncate to zero length or create text file for update
    a+   append; open or create text file for update, writing at end-of-file
    r+b *or* **rb+**   open binary file for update (reading and writing)
35 **w+b** *or* **wb+**   truncate to zero length or create binary file for update
    a+b *or* **ab+**   append; open or create binary file for update, writing at end-of-file

Opening a file with read mode ('**r**' as the first character in the **mode** argument) fails if the
file does not exist or cannot be read.

Opening a file with append mode ('**a**' as the first character in the **mode** argument) causes all
40 subsequent writes to the file to be forced to the then current end-of-file, regardless of intervening

_____
115. Additional characters may follow these sequences.

calls to the **fseek** function. In some implementations, opening a binary file with append mode
(**'b'** as the second or third character in the above list of **mode** argument values) may initially
position the file position indicator for the stream beyond the last data written, because of null
character padding.

5   When a file is opened with update mode (**'+'** as the second or third character in the above
list of **mode** argument values), both input and output may be performed on the associated stream.
However, output may not be directly followed by input without an intervening call to the
fflush function or to a file positioning function (**fseek**, **fsetpos**, or **rewind**), and input
may not be directly followed by output without an intervening call to a file positioning function,
10   unless the input operation encounters end-of-file. Opening (or creating) a text file with update
mode may instead open (or create) a binary stream in some implementations.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an
interactive device. The error and end-of-file indicators for the stream are cleared.

Returns

15   The **fopen** function returns a pointer to the object controlling the stream. If the open
operation fails, **fopen** returns a null pointer.

Forward references: file positioning functions (4.9.9).

4.9.5.4 The **freopen** Function

Synopsis

20
```
        #include <stdio.h>
        FILE *freopen(const char *filename, const char *mode,
               FILE *stream);
```

Description

The **freopen** function opens the file whose name is the string pointed to by **filename** and
25   associates the stream pointed to by **stream** with it. The **mode** argument is used just as in the
fopen function.[116]

The **freopen** function first attempts to close any file that is associated with the specified
stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for
the stream are cleared.

30   **Returns**

The **freopen** function returns a null pointer if the open operation fails. Otherwise,
freopen returns the value of **stream**.

4.9.5.5 The **setbuf** Function

Synopsis

35
```
        #include <stdio.h>
        void setbuf(FILE *stream, char *buf);
```

---

116. The primary use of the **freopen** function is to change the file associated with a standard text stream
     (**stderr**, **stdin**, or **stdout**), as those identifiers need not be modifiable lvalues to which the value returned
     by the **fopen** function may be assigned.

Description

Except that it returns no value, the **setbuf** function is equivalent to the **setvbuf** function invoked with the values **_IOFBF** for **mode** and **BUFSIZ** for **size**, or (if **buf** is a null pointer), with the value **_IONBF** for **mode**.

5   **Returns**

The **setbuf** function returns no value.

Forward references: the **setvbuf** function (4.9.5.6).

4.9.5.6 The setvbuf Function

Synopsis

10
```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Description

The **setvbuf** function may be used only after the stream pointed to by **stream** has been associated with an open file and before any other operation is performed on the stream. The
15 argument **mode** determines how **stream** will be buffered, as follows: **_IOFBF** causes input/output to be fully buffered; **_IOLBF** causes input/output to be line buffered; **_IONBF** causes input/output to be unbuffered. If **buf** is not a null pointer, the array it points to may be used instead of a buffer allocated by the **setvbuf** function.[117] The argument **size** specifies the size of the array. The contents of the array at any time are indeterminate.

20 **Returns**

The **setvbuf** function returns zero on success, or nonzero if an invalid value is given for **mode** or if the request cannot be honored.

4.9.6 Formatted Input/Output Functions

4.9.6.1 The fprintf Function

25 **Synopsis**

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

Description

The **fprintf** function writes output to the stream pointed to by **stream**, under control of
30 the string pointed to by **format** that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **fprintf** function returns when the end of the format string is encountered.

35   The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. After the %, the following appear in sequence:

---

117. The buffer must have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit.

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.

- An optional minimum *field width*. If the converted value has fewer characters than the field width, it will be padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an
5 asterisk * (described later) or a decimal integer.[118]

- An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be written from a string in **s** conversion. The
10 precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.

- An optional **h** specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will have been promoted
15 according to the integral promotions, and its value shall be converted to **short int** or **unsigned short int** before printing); an optional **h** specifying that a following **n** conversion specifier applies to a pointer to a **short int** argument; an optional **l** (ell) specifying that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; an optional **l** specifying that a following **n** conversion
20 specifier applies to a pointer to a **long int** argument; or an optional **L** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion specifier applies to a **long double** argument. If an **h**, **l**, or **L** appears with any other conversion specifier, the behavior is undefined.

- A character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this
25 case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a − flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are

30 −   The result of the conversion will be left-justified within the field. (It will be right-justified if this flag is not specified.)

+   The result of a signed conversion will always begin with a plus or minus sign. (It will begin with a sign only when a negative value is converted if this flag is not specified.)

space If the first character of a signed conversion is not a sign, or if a signed conversion results
35   in no characters, a space will be prefixed to the result. If the *space* and + flags both appear, the *space* flag will be ignored.

The result is to be converted to an "alternate form." For **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (or **X**) conversion, a nonzero result will have **0x** (or **0X**) prefixed to it. For **e**, **E**, **f**, **g**, and **G** conversions, the
40   result will always contain a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For **g** and **G** conversions, trailing zeros will *not* be removed from the result. For other conversions, the behavior is undefined.

---

118. Note that **0** is taken as a flag, not as the beginning of a field width.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and − flags both appear, the **0** flag will be ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag will be ignored. For other conversions, the behavior is undefined.

The conversion specifiers and their meanings are

d,i The **int** argument is converted to signed decimal in the style *[−]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o,u,x,X The **unsigned int** argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style *dddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f The **double** argument is converted to decimal notation in the style *[−]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

e,E The **double** argument is converted in the style *[−]d.ddd**e**±dd*, where there is one digit before the decimal-point character (which is nonzero if the argument is nonzero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

g,G The **double** argument is converted in style **f** or **e** (or in style **E** in the case of a **G** conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style **e** (or **E**) will be used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point character appears only if it is followed by a digit.

c The **int** argument is converted to an **unsigned char**, and the resulting character is written.

s The argument shall be a pointer to an array of character type.[119] Characters from the array are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

---

119. No special provisions are made for multibyte characters.

p   The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

n   The argument shall be a pointer to an integer into which is *written* the number of characters written to the output stream so far by this call to **fprintf**. No argument
5   is converted.

%   A **%** is written. No argument is converted. The complete conversion specification shall be **%%**.

If a conversion specification is invalid, the behavior is undefined.[120]

If any argument is, or points to, a union or an aggregate (except for an array of character type
10   using **%s** conversion, or a pointer using **%p** conversion), the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

Returns

The **fprintf** function returns the number of characters transmitted, or a negative value if an
15   output error occurred.

Environmental Limit

The minimum value for the maximum number of characters produced by any single conversion shall be 509.

Examples

20   To print a date and time in the form ``Sunday, July 3, 10:02'' followed by π to five decimal places:

```
#include <math.h>
#include <stdio.h>
/*...*/
char *weekday, *month;        /* pointers to strings */
int day, hour, min;
fprintf(stdout, "%s, %s %d, %.2d:%.2d\n",
        weekday, month, day, hour, min);
fprintf(stdout, "pi = %.5f\n", 4 * atan(1.0));
```

30 **4.9.6.2 The fscanf Function**

Synopsis

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Description

35   The **fscanf** function reads input from the stream pointed to by **stream**, under control of the string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as
40   always) but are otherwise ignored.

---

120. See ``future library directions'' (4.13.6).

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither `%` nor a white-space character); or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear
5 in sequence:

 • An optional assignment-suppressing character `*`.

 • An optional nonzero decimal integer that specifies the maximum field width.

 • An optional **h**, **l** (ell) or **L** indicating the size of the receiving object. The conversion specifiers **d**, **i**, and **n** shall be preceded by **h** if the corresponding argument is a pointer to
10   **short int** rather than a pointer to **int**, or by **l** if it is a pointer to **long int**. Similarly, the conversion specifiers **o**, **u**, and **x** shall be preceded by **h** if the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by **l** if it is a pointer to **unsigned long int**. Finally, the conversion specifiers **e**, **f**, and **g** shall be preceded by **l** if the corresponding argument is a pointer to **double** rather than a pointer to
15   **float**, or by **L** if it is a pointer to **long double**. If an **h**, **l**, or **L** appears with any other conversion specifier, the behavior is undefined.

 • A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The **fscanf** function executes each directive of the format in turn. If a directive fails, as
20 detailed below, the **fscanf** function returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary multibyte character is executed by reading the next characters
25 of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the **isspace** function) are skipped, unless the
30 specification includes a **[**, **c**, or **n** specifier.[121]

An input item is read from the stream, unless the specification includes an **n** specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is
35 zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching
40 failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

---

121. These white-space characters are not counted against a specified field width.

The following conversion specifiers are valid:

d   Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to integer.

5 **i**   Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the **strtol** function with the value 0 for the **base** argument. The corresponding argument shall be a pointer to integer.

o   Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 8 for the **base** argument.
10   The corresponding argument shall be a pointer to unsigned integer.

u   Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the **strtoul** function with the value 10 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

x   Matches an optionally signed hexadecimal integer, whose format is the same as expected
15   for the subject sequence of the **strtoul** function with the value 16 for the **base** argument. The corresponding argument shall be a pointer to unsigned integer.

e,f,g   Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the **strtod** function. The corresponding argument shall be a pointer to floating.

20 **s**   Matches a sequence of non-white-space characters.[122] The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

[Matches a nonempty sequence of characters[122] from a set of expected characters (the *scanset*). The corresponding argument shall be a pointer to the initial character of an
25   array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the **format** string, up to and including the matching right bracket (**]**). The characters between the brackets (the *scanlist*) comprise the scanset, unless the character after the left bracket is a circumflex (**^**), in which case the scanset contains all characters that do
30   not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with **[]** or **[^]**, the right bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise the first right bracket character is the one that ends the specification. If a **−** character is in the scanlist and is not the first, nor the second where the first character is
35   a **^**, nor the last character, the behavior is implementation-defined.

c   Matches a sequence of characters[122] of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

40 **p**   Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the **%p** conversion of the **fprintf** function. The corresponding argument shall be a pointer to a pointer to **void**. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to
45   that value; otherwise the behavior of the **%p** conversion is undefined.

---

122. No special provisions are made for multibyte characters.

n   No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the **fscanf** function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the **fscanf** function.

5   **%**   Matches a single %; no conversion or assignment occurs. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behavior is undefined.[123]

The conversion specifiers **E**, **G**, and **X** are also valid and behave the same as, respectively, **e**, **g**, and **x**.

10   If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

15   If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

Returns

20   The **fscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **fscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

Examples

The call:

```
25        #include <stdio.h>
          /*...*/
          int n, i; float x; char name[50];
          n = fscanf(stdin, "%d%f%s", &i, &x, name);
```

with the input line:

30   25 54.32E-1 thompson

will assign to *n* the value 3, to *i* the value 25, to *x* the value 5.432, and *name* will contain **thompson\0**. Or:

```
          #include <stdio.h>
          /*...*/
35        int i; float x; char name[50];
          fscanf(stdin, "%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

          56789 0123 56a72

will assign to *i* the value 56 and to *x* the value 789.0, will skip 0123, and *name* will contain
40 **56\0**. The next character read from the input stream will be **a**.

---

123. See "future library directions" (4.13.6).

To accept repeatedly from **stdin** a quantity, a unit of measure and an item name:

```
#include <stdio.h>
/*...*/
int count; float quant; char units[21], item[21];
while (!feof(stdin) && !ferror(stdin)) {
        count = fscanf(stdin, "%f%20s of %20s",
                &quant, units, item);
        fscanf(stdin,"%*[^\n]");
}
```

If the **stdin** stream contains the following lines:

```
2 quarts of oil
-12.8degrees Celsius
lots of luck
10.0LBS       of
fertilizer
100ergs of energy
```

the execution of the above example will be analogous to the following assignments:

```
quant = 2; strcpy(units, "quarts"); strcpy(item, "oil");
count = 3;
quant = -12.8; strcpy(units, "degrees");
count = 2; /* "C" fails to match "o" */
count = 0; /* "l" fails to match "%f" */
quant = 10.0; strcpy(units, "LBS"); strcpy(item, "fertilizer");
count = 3;
count = 0; /* "100e" fails to match "%f" */
count = EOF;
```

Forward references: the **strtod** function (4.10.1.4), the **strtol** function (4.10.1.5), the **strtoul** function (4.10.1.6).

4.9.6.3 The `printf` Function

Synopsis

```
#include <stdio.h>
int printf(const char *format, ...);
```

Description

The **printf** function is equivalent to **fprintf** with the argument **stdout** interposed before the arguments to **printf**.

Returns

The **printf** function returns the number of characters transmitted, or a negative value if an output error occurred.

4.9.6.4 The `scanf` Function

Synopsis

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Description

The **scanf** function is equivalent to **fscanf** with the argument **stdin** interposed before the arguments to **scanf**.

Returns

The **scanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **scanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

5 **4.9.6.5 The sprintf Function**

Synopsis

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

Description

10   The **sprintf** function is equivalent to **fprintf**, except that the argument **s** specifies an array into which the generated output is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined.

Returns

15   The **sprintf** function returns the number of characters written in the array, not counting the terminating null character.

4.9.6.6 The sscanf Function

Synopsis

```
#include <stdio.h>
20     int sscanf(const char *s, const char *format, ...);
```

Description

The **sscanf** function is equivalent to **fscanf**, except that the argument **s** specifies a string from which the input is to be obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the **fscanf** function. If copying takes place 25   between objects that overlap, the behavior is undefined.

Returns

The **sscanf** function returns the value of the macro **EOF** if an input failure occurs before any conversion. Otherwise, the **sscanf** function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

30 **4.9.6.7 The vfprintf Function**

Synopsis

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg);
```

35 **Description**

The **vfprintf** function is equivalent to **fprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vfprintf** function does not invoke the **va_end** macro.[124]

---

124. As the functions **vfprintf**, **vsprintf**, and **vprintf** invoke the **va_arg** macro, the value of **arg** after the return is indeterminate.

Returns

The **vfprintf** function returns the number of characters transmitted, or a negative value if an output error occurred.

Example

5   The following shows the use of the **vfprintf** function in a general error-reporting routine.

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

void error(char *function_name, char *format, ...)
{
        va_list args;

        va_start(args, format);
        /* print out name of function causing error */
        fprintf(stderr, "ERROR in %s: ", function_name);
        /* print out remainder of message */
        vfprintf(stderr, format, args);
        va_end(args);
}
```

4.9.6.8 The **vprintf** Function

Synopsis

20
```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

Description

The **vprintf** function is equivalent to **printf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vprintf** function does not invoke the **va_end** macro.[124]

Returns

The **vprintf** function returns the number of characters transmitted, or a negative value if an output error occurred.

30 ### 4.9.6.9 The **vsprintf** Function

Synopsis

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

35 **Description**

The **vsprintf** function is equivalent to **sprintf**, with the variable argument list replaced by **arg**, which shall have been initialized by the **va_start** macro (and possibly subsequent **va_arg** calls). The **vsprintf** function does not invoke the **va_end** macro.[124] If copying takes place between objects that overlap, the behavior is undefined.

40 **Returns**

The **vsprintf** function returns the number of characters written in the array, not counting the terminating null character.

4.9.7 Character Input/Output Functions

4.9.7.1 The **fgetc** Function

Synopsis

```
        #include <stdio.h>
5       int fgetc(FILE *stream);
```

Description

The **fgetc** function obtains the next character (if present) as an **unsigned char** converted to an **int**, from the input stream pointed to by **stream**, and advances the associated file position indicator for the stream (if defined).

10   **Returns**

The **fgetc** function returns the next character from the input stream pointed to by **stream**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **fgetc** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **fgetc** returns **EOF**.[125]

4.9.7.2 The **fgets** Function

15   **Synopsis**

```
        #include <stdio.h>
        char *fgets(char *s, int n, FILE *stream);
```

Description

The **fgets** function reads at most one less than the number of characters specified by **n** from
20   the stream pointed to by **stream** into the array pointed to by **s**. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Returns

The **fgets** function returns **s** if successful. If end-of-file is encountered and no characters
25   have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

4.9.7.3 The **fputc** Function

Synopsis

```
30      #include <stdio.h>
        int fputc(int c, FILE *stream);
```

Description

The **fputc** function writes the character specified by **c** (converted to an **unsigned char**) to the output stream pointed to by **stream**, at the position indicated by the associated file
35   position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

---

125. An end-of-file and a read error can be distinguished by use of the **feof** and **ferror** functions.

Returns

The **fputc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **fputc** returns **EOF**.

4.9.7.4 The fputs Function

5   **Synopsis**

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Description

The **fputs** function writes the string pointed to by **s** to the stream pointed to by **stream**.
10   The terminating null character is not written.

Returns

The **fputs** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative value.

4.9.7.5 The getc Function

15   **Synopsis**

```
#include <stdio.h>
int getc(FILE *stream);
```

Description

The **getc** function is equivalent to **fgetc**, except that if it is implemented as a macro, it
20   may evaluate **stream** more than once, so the argument should never be an expression with side effects.

Returns

The **getc** function returns the next character from the input stream pointed to by **stream**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and **getc** returns
25   **EOF**. If a read error occurs, the error indicator for the stream is set and **getc** returns **EOF**.

4.9.7.6 The getchar Function

Synopsis

```
#include <stdio.h>
int getchar(void);
```

30   **Description**

The **getchar** function is equivalent to **getc** with the argument **stdin**.

Returns

The **getchar** function returns the next character from the input stream pointed to by **stdin**. If the stream is at end-of-file, the end-of-file indicator for the stream is set and
35   **getchar** returns **EOF**. If a read error occurs, the error indicator for the stream is set and **getchar** returns **EOF**.

4.9.7.7 The gets Function

Synopsis

```
#include <stdio.h>
40          char *gets(char *s);
```

Description

The **gets** function reads characters from the input stream pointed to by **stdin**, into the array pointed to by **s**, until end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read
5 into the array.

Returns

The **gets** function returns **s** if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a
10 null pointer is returned.

4.9.7.8 The **putc** Function

Synopsis

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

15 **Description**

The **putc** function is equivalent to **fputc**, except that if it is implemented as a macro, it may evaluate **stream** more than once, so the argument should never be an expression with side effects.

Returns

20 The **putc** function returns the character written. If a write error occurs, the error indicator for the stream is set and **putc** returns **EOF**.

4.9.7.9 The **putchar** Function

Synopsis

```
#include <stdio.h>
25      int putchar(int c);
```

Description

The **putchar** function is equivalent to **putc** with the second argument **stdout**.

Returns

The **putchar** function returns the character written. If a write error occurs, the error
30 indicator for the stream is set and **putchar** returns **EOF**.

4.9.7.10 The **puts** Function

Synopsis

```
#include <stdio.h>
int puts(const char *s);
```

35 **Description**

The **puts** function writes the string pointed to by **s** to the stream pointed to by **stdout**, and appends a new-line character to the output. The terminating null character is not written.

Returns

The **puts** function returns **EOF** if a write error occurs; otherwise it returns a nonnegative
40 value.

4.9.7.11 The ungetc Function

Synopsis

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

5 **Description**

The **ungetc** function pushes the character specified by **c** (converted to an **unsigned char**) back onto the input stream pointed to by **stream**. The pushed-back characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by **stream**) to a file positioning function (**fseek**,
10 **fsetpos**, or **rewind**) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of pushback is guaranteed. If the **ungetc** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

15 If the value of **c** equals that of the macro **EOF**, the operation fails and the input stream is unchanged.

A successful call to the **ungetc** function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. For a text stream,
20 the value of its file position indicator after a successful call to the **ungetc** function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the **ungetc** function; if its value was zero before a call, it is indeterminate after the call.

Returns

25 The **ungetc** function returns the character pushed back after conversion, or **EOF** if the operation fails.

Forward references: file positioning functions (4.9.9).

4.9.8 Direct Input/Output Functions

4.9.8.1 The fread Function

30 **Synopsis**

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb,
        FILE *stream);
```

Description

35 The **fread** function reads, into the array pointed to by **ptr**, up to **nmemb** elements whose size is specified by **size**, from the stream pointed to by **stream**. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

40 **Returns**

The **fread** function returns the number of elements successfully read, which may be less than **nmemb** if a read error or end-of-file is encountered. If **size** or **nmemb** is zero, **fread** returns zero and the contents of the array and the state of the stream remain unchanged.

4.9.8.2 The **fwrite** Function

Synopsis

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
        FILE *stream);
```

Description

The **fwrite** function writes, from the array pointed to by **ptr**, up to **nmemb** elements whose size is specified by **size**, to the stream pointed to by **stream**. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate.

Returns

The **fwrite** function returns the number of elements successfully written, which will be less than **nmemb** only if a write error is encountered.

4.9.9 File Positioning Functions

4.9.9.1 The **fgetpos** Function

Synopsis

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Description

The **fgetpos** function stores the current value of the file position indicator for the stream pointed to by **stream** in the object pointed to by **pos**. The value stored contains unspecified information usable by the **fsetpos** function for repositioning the stream to its position at the time of the call to the **fgetpos** function.

Returns

If successful, the **fgetpos** function returns zero; on failure, the **fgetpos** function returns nonzero and stores an implementation-defined positive value in **errno**.

Forward references: the **fsetpos** function (4.9.9.3).

4.9.9.2 The **fseek** Function

Synopsis

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

Description

The **fseek** function sets the file position indicator for the stream pointed to by **stream**.

For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding **offset** to the position specified by **whence**. The specified position is the beginning of the file if **whence** is **SEEK_SET**, the current value of the file position indicator if **SEEK_CUR**, or end-of-file if **SEEK_END**. A binary stream need not meaningfully support **fseek** calls with a **whence** value of **SEEK_END**.

For a text stream, either **offset** shall be zero, or **offset** shall be a value returned by an earlier call to the **ftell** function on the same stream and **whence** shall be **SEEK_SET**.

A successful call to the **fseek** function clears the end-of-file indicator for the stream and undoes any effects of the **ungetc** function on the same stream. After an **fseek** call, the next operation on an update stream may be either input or output.

Returns

The **fseek** function returns nonzero only for a request that cannot be satisfied.

Forward references: the **ftell** function (4.9.9.4).

4.9.9.3 The **fsetpos** Function

5 **Synopsis**

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description

The **fsetpos** function sets the file position indicator for the stream pointed to by **stream**
10 according to the value of the object pointed to by **pos**, which shall be a value obtained from an
earlier call to the **fgetpos** function on the same stream.

A successful call to the **fsetpos** function clears the end-of-file indicator for the stream and
undoes any effects of the **ungetc** function on the same stream. After an **fsetpos** call, the
next operation on an update stream may be either input or output.

15 **Returns**

If successful, the **fsetpos** function returns zero; on failure, the **fsetpos** function returns
nonzero and stores an implementation-defined positive value in **errno**.

4.9.9.4 The **ftell** Function

Synopsis

20
```
#include <stdio.h>
long int ftell(FILE *stream);
```

Description

The **ftell** function obtains the current value of the file position indicator for the stream
pointed to by **stream**. For a binary stream, the value is the number of characters from the
25 beginning of the file. For a text stream, its file position indicator contains unspecified
information, usable by the **fseek** function for returning the file position indicator for the stream
to its position at the time of the **ftell** call; the difference between two such return values is not
necessarily a meaningful measure of the number of characters written or read.

Returns

30 If successful, the **ftell** function returns the current value of the file position indicator for
the stream. On failure, the **ftell** function returns −1L and stores an implementation-defined
positive value in **errno**.

4.9.9.5 The **rewind** Function

Synopsis

35
```
#include <stdio.h>
void rewind(FILE *stream);
```

Description

The **rewind** function sets the file position indicator for the stream pointed to by **stream** to
the beginning of the file. It is equivalent to

40
```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

Returns

The **rewind** function returns no value.

4.9.10 Error-Handling Functions

4.9.10.1 The **clearerr** Function

5 Synopsis

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Description

The **clearerr** function clears the end-of-file and error indicators for the stream pointed to
10 by **stream**.

Returns

The **clearerr** function returns no value.

4.9.10.2 The **feof** Function

Synopsis

15 ```
#include <stdio.h>
int feof(FILE *stream);
```

Description

The **feof** function tests the end-of-file indicator for the stream pointed to by **stream**.

Returns

20 The **feof** function returns nonzero if and only if the end-of-file indicator is set for **stream**.

4.9.10.3 The **ferror** Function

Synopsis

```
#include <stdio.h>
int ferror(FILE *stream);
```

25 **Description**

The **ferror** function tests the error indicator for the stream pointed to by **stream**.

Returns

The **ferror** function returns nonzero if and only if the error indicator is set for **stream**.

4.9.10.4 The **perror** Function

30 Synopsis

```
#include <stdio.h>
void perror(const char *s);
```

Description

The **perror** function maps the error number in the integer expression **errno** to an error
35 message. It writes a sequence of characters to the standard error stream thus: first (if **s** is not a
null pointer and the character pointed to by **s** is not the null character), the string pointed to by **s**
followed by a colon (:) and a space; then an appropriate error message string followed by a
new-line character. The contents of the error message strings are the same as those returned by
the **strerror** function with argument **errno**, which are implementation-defined.

Returns

The **perror** function returns no value.

Forward references: the **strerror** function (4.11.6.2).

4.10 General Utilities **<stdlib.h>**

The header **<stdlib.h>** declares four types and several functions of general utility, and defines several macros.[126]

The types declared are **size_t** and **wchar_t** (both described in 4.1.5),

5   **div_t**

which is a structure type that is the type of the value returned by the **div** function, and

ldiv_t

which is a structure type that is the type of the value returned by the **ldiv** function.

The macros defined are **NULL** (described in 4.1.5);

10   **EXIT_FAILURE**

and

EXIT_SUCCESS

which expand to integral expressions that may be used as the argument to the **exit** function to return unsuccessful or successful termination status, respectively, to the host environment;

15   **RAND_MAX**

which expands to an integral constant expression, the value of which is the maximum value returned by the **rand** function; and

MB_CUR_MAX

which expands to a positive integer expression whose value is the maximum number of bytes in a
20 multibyte character for the extended character set specified by the current locale (category **LC_CTYPE**), and whose value is never greater than **MB_LEN_MAX**.

4.10.1 String Conversion Functions

The functions **atof**, **atoi**, and **atol** need not affect the value of the integer expression **errno** on an error. If the value of the result cannot be represented, the behavior is undefined.

25 **4.10.1.1 The atof Function**

Synopsis

```
#include <stdlib.h>
double atof(const char *nptr);
```

Description

30   The **atof** function converts the initial portion of the string pointed to by **nptr** to **double** representation. Except for the behavior on error, it is equivalent to

```
strtod(nptr, (char **)NULL)
```

Returns

The **atof** function returns the converted value.

35 **Forward references:** the **strtod** function (4.10.1.4).

---

126. See ``future library directions'' (4.13.7).

4.10.1.2 The atoi Function

Synopsis

```
#include <stdlib.h>
int atoi(const char *nptr);
```

5 **Description**

The **atoi** function converts the initial portion of the string pointed to by **nptr** to **int** representation. Except for the behavior on error, it is equivalent to

```
(int)strtol(nptr, (char **)NULL, 10)
```

Returns

10   The **atoi** function returns the converted value.

Forward references: the **strtol** function (4.10.1.5).

4.10.1.3 The atol Function

Synopsis

```
#include <stdlib.h>
long int atol(const char *nptr);
```

Description

The **atol** function converts the initial portion of the string pointed to by **nptr** to **long int** representation. Except for the behavior on error, it is equivalent to

```
strtol(nptr, (char **)NULL, 10)
```

20 **Returns**

The **atol** function returns the converted value.

Forward references: the **strtol** function (4.10.1.5).

4.10.1.4 The strtod Function

Synopsis

25
```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Description

The **strtod** function converts the initial portion of the string pointed to by **nptr** to **double** representation. First, it decomposes the input string into three parts: an initial, possibly
30 empty, sequence of white-space characters (as specified by the **isspace** function), a subject sequence resembling a floating-point constant; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then a nonempty
35 sequence of digits optionally containing a decimal-point character, then an optional exponent part as defined in 3.1.3.1, but no floating suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a
40 digit, or a decimal-point character.

If the subject sequence has the expected form, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) is interpreted as a floating constant according to the rules of 3.1.3.1, except that the decimal-point character is used in place

of a period, and that if neither an exponent part nor a decimal-point character appears, a decimal point is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

5   In other than the **"C"** locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

10   **Returns**

The **strtod** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus **HUGE_VAL** is returned (according to the sign of the value), and the value of the macro **ERANGE** is stored in **errno**. If the correct value would cause underflow, zero is returned and
15   the value of the macro **ERANGE** is stored in **errno**.

4.10.1.5 The **strtol** Function

Synopsis

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

20   **Description**

The **strtol** function converts the initial portion of the string pointed to by **nptr** to **long int** representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **isspace** function), a subject sequence resembling an integer represented in some radix determined by the value of **base**, and
25   a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to an integer, and returns the result.

If the value of **base** is zero, the expected form of the subject sequence is that of an integer constant as described in 3.1.3.2, optionally preceded by a plus or minus sign, but not including an
30   integer suffix. If the value of **base** is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by **base**, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of **base** are permitted. If the value of **base** is 16, the
35   characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the
40   first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules of 3.1.3.2. If the subject sequence has the expected form and the value of **base** is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the
45   subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

In other than the "**C**" locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that
5 **endptr** is not a null pointer.

Returns

The **strtol** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **LONG_MAX** or **LONG_MIN** is returned (according to the sign of the value), and the value of the
10 macro **ERANGE** is stored in **errno**.

4.10.1.6 The **strtoul** Function

Synopsis

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr,
15        int base);
```

Description

The **strtoul** function converts the initial portion of the string pointed to by **nptr** to **unsigned long int** representation. First, it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the **isspace**
20 function), a subject sequence resembling an unsigned integer represented in some radix determined by the value of **base**, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then, it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of **base** is zero, the expected form of the subject sequence is that of an integer
25 constant as described in 3.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of **base** is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by **base**, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from **a** (or **A**) through **z** (or **Z**) are ascribed the values 10 to 35; only letters whose
30 ascribed values are less than that of **base** are permitted. If the value of **base** is 16, the characters **0x** or **0X** may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence
35 contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of **base** is zero, the sequence of characters starting with the first digit is interpreted as an integer constant according to the rules of 3.1.3.2. If the subject sequence has the expected form and the value of **base** is between 2 and
40 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

In other than the "**C**" locale, additional implementation-defined subject sequence forms may
45 be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of **nptr** is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

Returns

The **strtoul** function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, **ULONG_MAX** is returned, and the value of the macro **ERANGE** is stored in **errno**.

5 4.10.2 Pseudo-Random Sequence Generation Functions

4.10.2.1 The **rand** Function

Synopsis

```
#include <stdlib.h>
int rand(void);
```

10 Description

The **rand** function computes a sequence of pseudo-random integers in the range 0 to **RAND_MAX**.

The implementation shall behave as if no library function calls the **rand** function.

Returns

15 The **rand** function returns a pseudo-random integer.

Environmental Limit

The value of the **RAND_MAX** macro shall be at least 32767.

4.10.2.2 The **srand** Function

Synopsis

```
20     #include <stdlib.h>
       void srand(unsigned int seed);
```

Description

The **srand** function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to **rand**. If **srand** is then called with the same 25 seed value, the sequence of pseudo-random numbers shall be repeated. If **rand** is called before any calls to **srand** have been made, the same sequence shall be generated as when **srand** is first called with a seed value of 1.

The implementation shall behave as if no library function calls the **srand** function.

Returns

30 The **srand** function returns no value.

Example

The following functions define a portable implementation of **rand** and **srand**.

```
        static unsigned long int next = 1;

        int rand(void)     /* RAND_MAX assumed to be 32767 */
        {
                next = next * 1103515245 + 12345;
                return (unsigned int)(next/65536) % 32768;
        }

        void srand(unsigned int seed)
        {
                next = seed;
        }
```

4.10.3 Memory Management Functions

The order and contiguity of storage allocated by successive calls to the **calloc**, **malloc**, and **realloc** functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object or an array of such objects in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined; the value returned shall be either a null pointer or a unique pointer. The value of a pointer that refers to freed space is indeterminate.

4.10.3.1 The **calloc** Function

Synopsis

```
        #include <stdlib.h>
        void *calloc(size_t nmemb, size_t size);
```

Description

The **calloc** function allocates space for an array of **nmemb** objects, each of whose size is **size**. The space is initialized to all bits zero.[127]

Returns

The **calloc** function returns either a null pointer or a pointer to the allocated space.

4.10.3.2 The **free** Function

Synopsis

```
        #include <stdlib.h>
        void free(void *ptr);
```

Description

The **free** function causes the space pointed to by **ptr** to be deallocated, that is, made available for further allocation. If **ptr** is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the **calloc**, **malloc**, or **realloc** function, or if the space has been deallocated by a call to **free** or **realloc**, the behavior is undefined.

---

127. Note that this need not be the same as the representation of floating-point zero or a null pointer constant.

Returns

The **free** function returns no value.

4.10.3.3 The **malloc** Function

Synopsis

5
```
#include <stdlib.h>
void *malloc(size_t size);
```

Description

The **malloc** function allocates space for an object whose size is specified by **size** and whose value is indeterminate.

10   **Returns**

The **malloc** function returns either a null pointer or a pointer to the allocated space.

4.10.3.4 The **realloc** Function

Synopsis

```
#include <stdlib.h>
```
15
```
void *realloc(void *ptr, size_t size);
```

Description

The **realloc** function changes the size of the object pointed to by **ptr** to the size specified by **size**. The contents of the object shall be unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is
20   indeterminate. If **ptr** is a null pointer, the **realloc** function behaves like the **malloc** function for the specified size. Otherwise, if **ptr** does not match a pointer earlier returned by the **calloc**, **malloc**, or **realloc** function, or if the space has been deallocated by a call to the **free** or **realloc** function, the behavior is undefined. If the space cannot be allocated, the object pointed to by **ptr** is unchanged. If **size** is zero and **ptr** is not a null pointer, the object
25   it points to is freed.

Returns

The **realloc** function returns either a null pointer or a pointer to the possibly moved allocated space.

4.10.4 Communication with the Environment

30 ### 4.10.4.1 The **abort** Function

Synopsis

```
#include <stdlib.h>
void abort(void);
```

Description

35   The **abort** function causes abnormal program termination to occur, unless the signal **SIGABRT** is being caught and the signal handler does not return. Whether open output streams are flushed or open streams closed or temporary files removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call **raise(SIGABRT)**.

40   **Returns**

The **abort** function cannot return to its caller.

4.10.4.2 The `atexit` Function

Synopsis

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

5 **Description**

The **atexit** function registers the function pointed to by **func**, to be called without arguments at normal program termination.

Implementation Limits

The implementation shall support the registration of at least 32 functions.

10 **Returns**

The **atexit** function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the **exit** function (4.10.4.3).

4.10.4.3 The `exit` Function

Synopsis

15
```
#include <stdlib.h>
void exit(int status);
```

Description

The **exit** function causes normal program termination to occur. If more than one call to the **exit** function is executed by a program, the behavior is undefined.

20 First, all functions registered by the **atexit** function are called, in the reverse order of their registration.[128]

Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the **tmpfile** function are removed.

Finally, control is returned to the host environment. If the value of **status** is zero or
25 **EXIT_SUCCESS**, an implementation-defined form of the status *successful termination* is returned. If the value of **status** is **EXIT_FAILURE**, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.

Returns

30 The **exit** function cannot return to its caller.

4.10.4.4 The `getenv` Function

Synopsis

```
#include <stdlib.h>
char *getenv(const char *name);
```

35 **Description**

The **getenv** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by **name**. The set of environment names and the method for altering the environment list are implementation-defined.

---

128. Each function is called as many times as it was registered.

The implementation shall behave as if no library function calls the **getenv** function.

Returns

The **getenv** function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a
5   subsequent call to the **getenv** function. If the specified **name** cannot be found, a null pointer is returned.

4.10.4.5 The **system** Function

Synopsis

```
#include <stdlib.h>
10      int system(const char *string);
```

Description

The **system** function passes the string pointed to by **string** to the host environment to be executed by a *command processor* in an implementation-defined manner. A null pointer may be used for **string** to inquire whether a command processor exists.

15 **Returns**

If the argument is a null pointer, the **system** function returns nonzero only if a command processor is available. If the argument is not a null pointer, the **system** function returns an implementation-defined value.

4.10.5 Searching and Sorting Utilities

20 ### 4.10.5.1 The **bsearch** Function

Synopsis

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
        size_t nmemb, size_t size,
25          int (*compar)(const void *, const void *));
```

Description

The **bsearch** function searches an array of **nmemb** objects, the initial element of which is pointed to by **base**, for an element that matches the object pointed to by **key**. The size of each element of the array is specified by **size**.

30   The comparison function pointed to by **compar** is called with two arguments that point to the **key** object and to an array element, in that order. The function shall return an integer less than, equal to, or greater than zero if the **key** object is considered, respectively, to be less than, to match, or to be greater than the array element. The array shall consist of: all the elements that compare less than, all the elements that compare equal to, and all the elements that compare
35 greater than the **key** object, in that order.[129]

Returns

The **bsearch** function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

---

129. In practice, the entire array is sorted according to the comparison function.

4.10.5.2 The qsort Function

Synopsis

```
        #include <stdlib.h>
        void qsort(void *base, size_t nmemb, size_t size,
5               int (*compar)(const void *, const void *));
```

Description

The **qsort** function sorts an array of **nmemb** objects, the initial element of which is pointed to by **base**. The size of each object is specified by **size**.

The contents of the array are sorted into ascending order according to a comparison function
10 pointed to by **compar**, which is called with two arguments that point to the objects being compared. The function shall return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two elements compare as equal, their order in the sorted array is unspecified.

Returns

15   The **qsort** function returns no value.

4.10.6 Integer Arithmetic Functions

4.10.6.1 The abs Function

Synopsis

```
        #include <stdlib.h>
20      int abs(int j);
```

Description

The **abs** function computes the absolute value of an integer **j**. If the result cannot be represented, the behavior is undefined.[130]

Returns

25   The **abs** function returns the absolute value.

4.10.6.2 The div Function

Synopsis

```
        #include <stdlib.h>
        div_t div(int numer, int denom);
```

30 **Description**

The **div** function computes the quotient and remainder of the division of the numerator **numer** by the denominator **denom**. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined; otherwise, **quot * denom + rem** shall equal **numer**.

35 **Returns**

The **div** function returns a structure of type **div_t**, comprising both the quotient and the remainder. The structure shall contain the following members, in either order:

---

130. In a two's complement representation, the absolute value of the most negative number cannot be represented.

```
int quot;    /* quotient */
int rem;     /* remainder */
```

4.10.6.3 The **labs** Function

Synopsis

5
```
#include <stdlib.h>
long int labs(long int j);
```

Description

The **labs** function is similar to the **abs** function, except that the argument and the returned value each have type **long int**.

10 ### 4.10.6.4 The **ldiv** Function

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Description

15   The **ldiv** function is similar to the **div** function, except that the arguments and the members of the returned structure (which has type **ldiv_t**) all have type **long int**.

4.10.7 Multibyte Character Functions

The behavior of the multibyte character functions is affected by the **LC_CTYPE** category of the current locale. For a state-dependent encoding, each function is placed into its initial state by
20 a call for which its character pointer argument, **s**, is a null pointer. Subsequent calls with **s** as other than a null pointer cause the internal state of the function to be altered as necessary. A call with **s** as a null pointer causes these functions to return a nonzero value if encodings have state dependency, and zero otherwise.[131] Changing the **LC_CTYPE** category causes the shift state of these functions to be indeterminate.

25 ### 4.10.7.1 The **mblen** Function

Synopsis

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Description

30   If **s** is not a null pointer, the **mblen** function determines the number of bytes contained in the multibyte character pointed to by **s**. Except that the shift state of the **mbtowc** function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, s, n);
```

The implementation shall behave as if no library function calls the **mblen** function.

35 Returns

If **s** is a null pointer, the **mblen** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If **s** is not a null pointer, the **mblen** function either returns 0 (if **s** points to the null character), or returns the

---

131. If the implementation employs special bytes to change the shift state, these bytes do not produce separate wide character codes, but are grouped with an adjacent multibyte character.

number of bytes that are contained in the multibyte character (if the next **n** or fewer bytes form a valid multibyte character), or returns −1 (if they do not form a valid multibyte character).

Forward references: the **mbtowc** function (4.10.7.2).

4.10.7.2 The **mbtowc** Function

5 **Synopsis**

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Description

If **s** is not a null pointer, the **mbtowc** function determines the number of bytes that are
10 contained in the multibyte character pointed to by **s**. It then determines the code for the value of type **wchar_t** that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero.) If the multibyte character is valid and **pwc** is not a null pointer, the **mbtowc** function stores the code in the object pointed to by **pwc**. At most **n** bytes of the array pointed to by **s** will be examined.

15   The implementation shall behave as if no library function calls the **mbtowc** function.

Returns

If **s** is a null pointer, the **mbtowc** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If **s** is not a null pointer, the **mbtowc** function either returns 0 (if **s** points to the null character), or returns the
20 number of bytes that are contained in the converted multibyte character (if the next **n** or fewer bytes form a valid multibyte character), or returns −1 (if they do not form a valid multibyte character).

In no case will the value returned be greater than **n** or the value of the **MB_CUR_MAX** macro.

4.10.7.3 The **wctomb** Function

25 **Synopsis**

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Description

The **wctomb** function determines the number of bytes needed to represent the multibyte
30 character corresponding to the code whose value is **wchar** (including any change in shift state). It stores the multibyte character representation in the array object pointed to by **s** (if **s** is not a null pointer). At most **MB_CUR_MAX** characters are stored. If the value of **wchar** is zero, the **wctomb** function is left in the initial shift state.

The implementation shall behave as if no library function calls the **wctomb** function.

35 **Returns**

If **s** is a null pointer, the **wctomb** function returns a nonzero or zero value, if multibyte character encodings, respectively, do or do not have state-dependent encodings. If **s** is not a null pointer, the **wctomb** function returns −1 if the value of **wchar** does not correspond to a valid multibyte character, or returns the number of bytes that are contained in the multibyte character
40 corresponding to the value of **wchar**.

In no case will the value returned be greater than the value of the **MB_CUR_MAX** macro.

4.10.8 Multibyte String Functions

The behavior of the multibyte string functions is affected by the **LC_CTYPE** category of the current locale.

4.10.8.1 The **mbstowcs** Function

5   **Synopsis**

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Description

The **mbstowcs** function converts a sequence of multibyte characters that begins in the initial
10 shift state from the array pointed to by **s** into a sequence of corresponding codes and stores not
more than **n** codes into the array pointed to by **pwcs**. No multibyte characters that follow a null
character (which is converted into a code with value zero) will be examined or converted. Each
multibyte character is converted as if by a call to the **mbtowc** function, except that the shift state
of the **mbtowc** function is not affected.

15   No more than **n** elements will be modified in the array pointed to by **pwcs**. If copying takes
place between objects that overlap, the behavior is undefined.

Returns

If an invalid multibyte character is encountered, the **mbstowcs** function returns
(size_t)-1. Otherwise, the **mbstowcs** function returns the number of array elements
20 modified, not including a terminating zero code, if any.[132]

4.10.8.2 The **wcstombs** Function

Synopsis

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

25 **Description**

The **wcstombs** function converts a sequence of codes that correspond to multibyte characters
from the array pointed to by **pwcs** into a sequence of multibyte characters that begins in the
initial shift state and stores these multibyte characters into the array pointed to by **s**, stopping if a
multibyte character would exceed the limit of **n** total bytes or if a null character is stored. Each
30 code is converted as if by a call to the **wctomb** function, except that the shift state of the
wctomb function is not affected.

No more than **n** bytes will be modified in the array pointed to by **s**. If copying takes place
between objects that overlap, the behavior is undefined.

Returns

35   If a code is encountered that does not correspond to a valid multibyte character, the
wcstombs function returns **(size_t)-1**. Otherwise, the **wcstombs** function returns the
number of bytes modified, not including a terminating null character, if any.[132]

---

132. The array will not be null- or zero-terminated if the value returned is **n**.

4.11 String Handling <string.h>

4.11.1 String Function Conventions

5 The header **<string.h>** declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as arrays of character type.[133] The type is **size_t** and the macro is **NULL** (both described in 4.1.5). Various methods are used for determining the lengths of the arrays, but in all cases a **char *** or **void *** argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

4.11.2 Copying Functions

10 #### 4.11.2.1 The memcpy Function

Synopsis

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

Description

15 The **memcpy** function copies **n** characters from the object pointed to by **s2** into the object pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

Returns

The **memcpy** function returns the value of **s1**.

4.11.2.2 The memmove Function

20 **Synopsis**

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Description

The **memmove** function copies **n** characters from the object pointed to by **s2** into the object
25 pointed to by **s1**. Copying takes place as if the **n** characters from the object pointed to by **s2** are first copied into a temporary array of **n** characters that does not overlap the objects pointed to by **s1** and **s2**, and then the **n** characters from the temporary array are copied into the object pointed to by **s1**.

Returns

30 The **memmove** function returns the value of **s1**.

4.11.2.3 The strcpy Function

Synopsis

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

35 **Description**

The **strcpy** function copies the string pointed to by **s2** (including the terminating null character) into the array pointed to by **s1**. If copying takes place between objects that overlap, the behavior is undefined.

---

133. See "future library directions" (4.13.8).

Returns

The **strcpy** function returns the value of **s1**.

4.11.2.4 The **strncpy** Function

Synopsis

5
```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

Description

The **strncpy** function copies not more than **n** characters (characters that follow a null character are not copied) from the array pointed to by **s2** to the array pointed to by **s1**.[134] If
10 copying takes place between objects that overlap, the behavior is undefined.

If the array pointed to by **s2** is a string that is shorter than **n** characters, null characters are appended to the copy in the array pointed to by **s1**, until **n** characters in all have been written.

Returns

The **strncpy** function returns the value of **s1**.

15 ## 4.11.3 Concatenation Functions

4.11.3.1 The **strcat** Function

Synopsis

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

20 **Description**

The **strcat** function appends a copy of the string pointed to by **s2** (including the terminating null character) to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. If copying takes place between objects that overlap, the behavior is undefined.

25 **Returns**

The **strcat** function returns the value of **s1**.

4.11.3.2 The **strncat** Function

Synopsis

```
#include <string.h>
30      char *strncat(char *s1, const char *s2, size_t n);
```

Description

The **strncat** function appends not more than **n** characters (a null character and characters that follow it are not appended) from the array pointed to by **s2** to the end of the string pointed to by **s1**. The initial character of **s2** overwrites the null character at the end of **s1**. A
35 terminating null character is always appended to the result.[135] If copying takes place between objects that overlap, the behavior is undefined.

---

134. Thus, if there is no null character in the first **n** characters of the array pointed to by **s2**, the result will not be null-terminated.

135. Thus, the maximum number of characters that can end up in the array pointed to by **s1** is **strlen(s1)+n+1**.

Returns

The **strncat** function returns the value of **s1**.

Forward references: the **strlen** function (4.11.6.3).

4.11.4 Comparison Functions

5   The sign of a nonzero value returned by the comparison functions **memcmp**, **strcmp**, and **strncmp** is determined by the sign of the difference between the values of the first pair of characters (both interpreted as **unsigned char**) that differ in the objects being compared.

4.11.4.1 The memcmp Function

Synopsis

```
10        #include <string.h>
          int memcmp(const void *s1, const void *s2, size_t n);
```

Description

The **memcmp** function compares the first **n** characters of the object pointed to by **s1** to the first **n** characters of the object pointed to by **s2**.[136]

15 **Returns**

The **memcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by **s1** is greater than, equal to, or less than the object pointed to by **s2**.

4.11.4.2 The strcmp Function

Synopsis

```
20        #include <string.h>
          int strcmp(const char *s1, const char *s2);
```

Description

The **strcmp** function compares the string pointed to by **s1** to the string pointed to by **s2**.

Returns

25   The **strcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2**.

4.11.4.3 The strcoll Function

Synopsis

```
          #include <string.h>
30        int strcoll(const char *s1, const char *s2);
```

Description

The **strcoll** function compares the string pointed to by **s1** to the string pointed to by **s2**, both interpreted as appropriate to the **LC_COLLATE** category of the current locale.

---

136. The contents of ''holes'' used as padding for purposes of alignment within structure objects are indeterminate. Strings shorter than their allocated space and unions may also cause problems in comparison.

Returns

The **strcoll** function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by **s1** is greater than, equal to, or less than the string pointed to by **s2** when both are interpreted as appropriate to the current locale.

5 **4.11.4.4 The strncmp Function**

Synopsis

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Description

10   The **strncmp** function compares not more than **n** characters (characters that follow a null character are not compared) from the array pointed to by **s1** to the array pointed to by **s2**.

Returns

The **strncmp** function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by **s1** is greater than, equal to, or less than the
15 possibly null-terminated array pointed to by **s2**.

4.11.4.5 The strxfrm Function

Synopsis

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

20 **Description**

The **strxfrm** function transforms the string pointed to by **s2** and places the resulting string into the array pointed to by **s1**. The transformation is such that if the **strcmp** function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **strcoll** function applied to the same two original strings.
25 No more than **n** characters are placed into the resulting array pointed to by **s1**, including the terminating null character. If **n** is zero, **s1** is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Returns

The **strxfrm** function returns the length of the transformed string (not including the
30 terminating null character). If the value returned is **n** or more, the contents of the array pointed to by **s1** are indeterminate.

Example

The value of the following expression is the size of the array needed to hold the transformation of the string pointed to by **s**.

35   ` 1 + strxfrm(NULL, s, 0)`

4.11.5 Search Functions

4.11.5.1 The memchr Function

Synopsis

```
#include <string.h>
40      void *memchr(const void *s, int c, size_t n);
```

Description

The **memchr** function locates the first occurrence of **c** (converted to an **unsigned char**) in the initial **n** characters (each interpreted as **unsigned char**) of the object pointed to by **s**.

Returns

5   The **memchr** function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

4.11.5.2 The strchr Function

Synopsis

```
#include <string.h>
char *strchr(const char *s, int c);
```

Description

The **strchr** function locates the first occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

Returns

15   The **strchr** function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

4.11.5.3 The strcspn Function

Synopsis

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Description

The **strcspn** function computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters *not* from the string pointed to by **s2**.

Returns

25   The **strcspn** function returns the length of the segment.

4.11.5.4 The strpbrk Function

Synopsis

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

30   **Description**

The **strpbrk** function locates the first occurrence in the string pointed to by **s1** of any character from the string pointed to by **s2**.

Returns

The **strpbrk** function returns a pointer to the character, or a null pointer if no character
35   from **s2** occurs in **s1**.

4.11.5.5 The strrchr Function

Synopsis

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Description

The **strrchr** function locates the last occurrence of **c** (converted to a **char**) in the string pointed to by **s**. The terminating null character is considered to be part of the string.

Returns

5   The **strrchr** function returns a pointer to the character, or a null pointer if **c** does not occur in the string.

4.11.5.6 The **strspn** Function

Synopsis

```
         #include <string.h>
10       size_t strspn(const char *s1, const char *s2);
```

Description

The **strspn** function computes the length of the maximum initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

Returns

15   The **strspn** function returns the length of the segment.

4.11.5.7 The **strstr** Function

Synopsis

```
         #include <string.h>
         char *strstr(const char *s1, const char *s2);
```

20   **Description**

The **strstr** function locates the first occurrence in the string pointed to by **s1** of the sequence of characters (excluding the terminating null character) in the string pointed to by **s2**

Returns

The **strstr** function returns a pointer to the located string, or a null pointer if the string is
25   not found. If **s2** points to a string with zero length, the function returns **s1**.

4.11.5.8 The **strtok** Function

Synopsis

```
         #include <string.h>
         char *strtok(char *s1, const char *s2);
```

30   **Description**

A sequence of calls to the **strtok** function breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**. The first call in the sequence has **s1** as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by **s2** may be different from call
35   to call.

The first call in the sequence searches the string pointed to by **s1** for the first character that is *not* contained in the current separator string pointed to by **s2**. If no such character is found, then there are no tokens in the string pointed to by **s1** and the **strtok** function returns a null pointer. If such a character is found, it is the start of the first token.

40   The **strtok** function then searches from there for a character that *is* contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by **s1**, and subsequent searches for a token will return a null pointer. If such a character is found, it is overwritten by a null character, which terminates the current token. The

strtok function saves a pointer to the following character, from which the next search for a token will start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

5   The implementation shall behave as if no library function calls the **strtok** function.

Returns

The **strtok** function returns a pointer to the first character of a token, or a null pointer if there is no token.

Example

```
10        #include <string.h>
          static char str[] = "?a???b,,,#c";
          char *t;

          t = strtok(str, "?");    /* t points to the token "a" */
          t = strtok(NULL, ",");   /* t points to the token "??b" */
15        t = strtok(NULL, "#,");  /* t points to the token "c" */
          t = strtok(NULL, "?");   /* t is a null pointer */
```

4.11.6 Miscellaneous Functions

4.11.6.1 The memset Function

Synopsis

```
20        #include <string.h>
          void *memset(void *s, int c, size_t n);
```

Description

The **memset** function copies the value of **c** (converted to an **unsigned char**) into each of the first **n** characters of the object pointed to by **s**.

25 **Returns**

The **memset** function returns the value of **s**.

4.11.6.2 The strerror Function

Synopsis

```
          #include <string.h>
30        char *strerror(int errnum);
```

Description

The **strerror** function maps the error number in **errnum** to an error message string.

The implementation shall behave as if no library function calls the **strerror** function.

Returns

35   The **strerror** function returns a pointer to the string, the contents of which are implementation-defined. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **strerror** function.

4.11.6.3 The **strlen** Function

Synopsis

```
#include <string.h>
size_t strlen(const char *s);
```

5   Description

The **strlen** function computes the length of the string pointed to by **s**.

Returns

The **strlen** function returns the number of characters that precede the terminating null character.

4.12 Date and Time <time.h>

4.12.1 Components of Time

The header **<time.h>** defines two macros, and declares four types and several functions for
manipulating time. Many functions deal with a *calendar time* that represents the current date
5 (according to the Gregorian calendar) and time. Some functions deal with *local time*, which is
the calendar time expressed for some specific time zone, and with *Daylight Saving Time*, which
is a temporary change in the algorithm for determining local time. The local time zone and
Daylight Saving Time are implementation-defined.

The macros defined are **NULL** (described in 4.1.5); and

10   **CLOCKS_PER_SEC**

which is the number per second of the value returned by the **clock** function.

The types declared are **size_t** (described in 4.1.5);

           clock_t

and

15   **time_t**

which are arithmetic types capable of representing times; and

           struct tm

which holds the components of a calendar time, called the *broken-down time*. The structure shall
contain at least the following members, in any order. The semantics of the members and their
20 normal ranges are expressed in the comments.[137]

```
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 */
```

30 The value of **tm_isdst** is positive if Daylight Saving Time is in effect, zero if Daylight Saving
Time is not in effect, and negative if the information is not available.

4.12.2 Time Manipulation Functions

4.12.2.1 The clock Function

Synopsis

35   **#include <time.h>**
           clock_t clock(void);

Description

The **clock** function determines the processor time used.

---

137. The range [0, 61] for **tm_sec** allows for as many as two leap seconds.

Returns

The **clock** function returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the **clock** function should be divided by the value of the macro **CLOCKS_PER_SEC**. If the processor time used is not available or its value cannot be represented, the function returns the value **(clock_t)-1**.[138]

4.12.2.2 The **difftime** Function

Synopsis

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

Description

The **difftime** function computes the difference between two calendar times: **time1 - time0**.

Returns

The **difftime** function returns the difference expressed in seconds as a **double**.

4.12.2.3 The **mktime** Function

Synopsis

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Description

The **mktime** function converts the broken-down time, expressed as local time, in the structure pointed to by **timeptr** into a calendar time value with the same encoding as that of the values returned by the **time** function. The original values of the **tm_wday** and **tm_yday** components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above.[139] On successful completion, the values of the **tm_wday** and **tm_yday** components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of **tm_mday** is not set until **tm_mon** and **tm_year** are determined.

Returns

The **mktime** function returns the specified calendar time encoded as a value of type **time_t**. If the calendar time cannot be represented, the function returns the value **(time_t)-1**.

Example

What day of the week is July 4, 2001?

---

138. In order to measure the time spent in a program, the **clock** function should be called at the start of the program and its return value subtracted from the value returned by subsequent calls.

139. Thus, a positive or zero value for **tm_isdst** causes the **mktime** function to presume initially that Daylight Saving Time, respectively, is or is not in effect for the specified time. A negative value for **tm_isdst** causes the **mktime** function to attempt to determine whether Daylight Saving Time is in effect for the specified time.

```
          #include <stdio.h>
          #include <time.h>
          static const char *const wday[] = {
                  "Sunday", "Monday", "Tuesday", "Wednesday",
  5               "Thursday", "Friday", "Saturday", "-unknown-"
          };
          struct tm time_str;
          /*...*/

          time_str.tm_year    = 2001 - 1900;
 10       time_str.tm_mon     = 7 - 1;
          time_str.tm_mday    = 4;
          time_str.tm_hour    = 0;
          time_str.tm_min     = 0;
          time_str.tm_sec     = 1;
 15       time_str.tm_isdst   = -1;
          if (mktime(&time_str) == -1)
                  time_str.tm_wday = 7;
          printf("%s\n", wday[time_str.tm_wday]);
```

4.12.2.4 The time Function

20 Synopsis

```
          #include <time.h>
          time_t time(time_t *timer);
```

Description

The **time** function determines the current calendar time. The encoding of the value is
25 unspecified.

Returns

The **time** function returns the implementation's best approximation to the current calendar
time. The value **(time_t)-1** is returned if the calendar time is not available. If **timer** is not
a null pointer, the return value is also assigned to the object it points to.

30 ## 4.12.3 Time Conversion Functions

Except for the **strftime** function, these functions return values in one of two static objects:
a broken-down time structure and an array of **char**. Execution of any of the functions may
overwrite the information returned in either of these objects by any of the other functions. The
implementation shall behave as if no other library functions call these functions.

35 ### 4.12.3.1 The asctime Function

Synopsis

```
          #include <time.h>
          char *asctime(const struct tm *timeptr);
```

Description

40   The **asctime** function converts the broken-down time in the structure pointed to by
timeptr into a string in the form

```
          Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm.

```
       char *asctime(const struct tm *timeptr)
       {
               static const char wday_name[7][3] = {
                       "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
 5             };
               static const char mon_name[12][3] = {
                       "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                       "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
               };
10             static char result[26];

               sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
                       wday_name[timeptr->tm_wday],
                       mon_name[timeptr->tm_mon],
                       timeptr->tm_mday, timeptr->tm_hour,
15                     timeptr->tm_min, timeptr->tm_sec,
                       1900 + timeptr->tm_year);
               return result;
       }
```

Returns

20   The **asctime** function returns a pointer to the string.

4.12.3.2 The ctime Function

Synopsis

```
       #include <time.h>
       char *ctime(const time_t *timer);
```

25 **Description**

The **ctime** function converts the calendar time pointed to by **timer** to local time in the form of a string. It is equivalent to

```
       asctime(localtime(timer))
```

Returns

30   The **ctime** function returns the pointer returned by the **asctime** function with that broken-down time as argument.

Forward references: the **localtime** function (4.12.3.4).

4.12.3.3 The gmtime Function

Synopsis

35
```
       #include <time.h>
       struct tm *gmtime(const time_t *timer);
```

Description

The **gmtime** function converts the calendar time pointed to by **timer** into a broken-down time, expressed as Coordinated Universal Time (UTC).

40 **Returns**

The **gmtime** function returns a pointer to that object, or a null pointer if UTC is not available.

4.12.3.4 The **localtime** Function

Synopsis

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

5 **Description**

The **localtime** function converts the calendar time pointed to by **timer** into a broken-down time, expressed as local time.

Returns

The **localtime** function returns a pointer to that object.

10 ### 4.12.3.5 The **strftime** Function

Synopsis

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
        const char *format, const struct tm *timeptr);
```

15 **Description**

The **strftime** function places characters into the array pointed to by **s** as controlled by the string pointed to by **format**. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The **format** string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a **%** character followed by a
20 character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than **maxsize** characters are placed into the array. Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are
25 determined by the **LC_TIME** category of the current locale and by the values contained in the structure pointed to by **timeptr**.

%a   is replaced by the locale's abbreviated weekday name.
%A   is replaced by the locale's full weekday name.
%b   is replaced by the locale's abbreviated month name.
30 %B   is replaced by the locale's full month name.
%c   is replaced by the locale's appropriate date and time representation.
%d   is replaced by the day of the month as a decimal number (**01-31**).
%H   is replaced by the hour (24-hour clock) as a decimal number (**00-23**).
%I   is replaced by the hour (12-hour clock) as a decimal number (**01-12**).
35 %j   is replaced by the day of the year as a decimal number (**001-366**).
%m   is replaced by the month as a decimal number (**01-12**).
%M   is replaced by the minute as a decimal number (**00-59**).
%p   is replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock.
40 %S   is replaced by the second as a decimal number (**00-61**).
%U   is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (**00-53**).
%w   is replaced by the weekday as a decimal number (**0-6**), where Sunday is **0**.
%W   is replaced by the week number of the year (the first Monday as the first day of week 1)
45 as a decimal number (**00-53**).
%x   is replaced by the locale's appropriate date representation.
%X   is replaced by the locale's appropriate time representation.
%y   is replaced by the year without century as a decimal number (**00-99**).
%Y   is replaced by the year with century as a decimal number.

%Z   is replaced by the time zone name or abbreviation, or by no characters if no time zone is determinable.

%%   is replaced by **%**.

If a conversion specifier is not one of the above, the behavior is undefined.

5   **Returns**

If the total number of resulting characters including the terminating null character is not more than **maxsize**, the **strftime** function returns the number of characters placed into the array pointed to by **s** not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

4.13 Future Library Directions

The following names are grouped under individual headers for convenience. All external names described below are reserved no matter what headers are included by the program.

4.13.1 Errors **<errno.h>**

5   Macros that begin with **E** and a digit or **E** and an uppercase letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<errno.h>** header.

4.13.2 Character Handling **<ctype.h>**

Function names that begin with either **is** or **to**, and a lowercase letter (followed by any
10   combination of digits, letters, and underscore) may be added to the declarations in the **<ctype.h>** header.

4.13.3 Localization **<locale.h>**

Macros that begin with **LC_** and an uppercase letter (followed by any combination of digits, letters, and underscore) may be added to the definitions in the **<locale.h>** header.

15   ### 4.13.4 Mathematics **<math.h>**

The names of all existing functions declared in the **<math.h>** header, suffixed with **f** or **l**, are reserved respectively for corresponding functions with **float** and **long double** arguments and return values.

4.13.5 Signal Handling **<signal.h>**

20   Macros that begin with either **SIG** and an uppercase letter or **SIG_** and an uppercase letter (followed by any combination of digits, letters, and underscore) may be added to the definitions in the **<signal.h>** header.

4.13.6 Input/Output **<stdio.h>**

Lowercase letters may be added to the conversion specifiers in **fprintf** and **fscanf**.
25   Other characters may be used in extensions.

4.13.7 General Utilities **<stdlib.h>**

Function names that begin with **str** and a lowercase letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<stdlib.h>** header.

4.13.8 String Handling **<string.h>**

30   Function names that begin with **str**, **mem**, or **wcs** and a lowercase letter (followed by any combination of digits, letters, and underscore) may be added to the declarations in the **<string.h>** header.

Appendixes (These Appendixes are not part of American National Standard X3.159-1989, but are included for information only.)

These appendixes collect information that appears in the standard, and are not necessarily complete.

A. Language Syntax Summary

The notation is described in the introduction to Section 3 (Language).

A.1 Lexical Grammar

A.1.1 Tokens

(3.1) *token:*

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

(3.1) *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

A.1.2 Keywords

(3.1.1) *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

A.1.3 Identifiers

(3.1.2) *identifier:*

> *nondigit*
> *identifier nondigit*
> *identifier digit*

(3.1.2) *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
```

(3.1.2) *digit:* one of

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

A.1.4 Constants

(3.1.3) *constant:*

> *floating-constant*
> *integer-constant*
> *emmeration-constant*
> *character-constant*

(3.1.3.1) *floating-constant:*

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

(3.1.3.1) *fractional-constant:*

> *digit-sequence$_{opt}$. digit-sequence*
> *digit-sequence$_{opt}$.*

(3.1.3.1) *exponent-part:*

> **e** *sign$_{opt}$ digit-sequence*
> **E** *sign$_{opt}$ digit-sequence*

(3.1.3.1) *sign:* one of

> **+   -**

(3.1.3.1) *digit-sequence:*

> *digit*
> *digit-sequence digit*

(3.1.3.1) *floating-suffix:* one of

> **f   l   F   L**

(3.1.3.2) *integer-constant:*

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

(3.1.3.2) *decimal-constant:*

> *nonzero-digit*
> *decimal-constant digit*

(3.1.3.2) *octal-constant:*

> **0**
> *octal-constant octal-digit*

(3.1.3.2) *hexadecimal-constant:*

> **0x** *hexadecimal-digit*
> **0X** *hexadecimal-digit*
> *hexadecimal-constant hexadecimal-digit*

(3.1.3.2) *nonzero-digit:* one of

> **1   2   3   4   5   6   7   8   9**

(3.1.3.2) *octal-digit:* one of

> **0   1   2   3   4   5   6   7**

(3.1.3.2) *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**

(3.1.3.2) *integer-suffix:*

    unsigned-suffix long-suffix$_{opt}$
    long-suffix unsigned-suffix$_{opt}$

(3.1.3.2) *unsigned-suffix:* one of

    `u  U`

(3.1.3.2) *long-suffix:* one of

    `l  L`

(3.1.3.3) *enumeration-constant:*

    identifier

(3.1.3.4) *character-constant:*

    `'` *c-char-sequence* `'`
    `L'` *c-char-sequence* `'`

(3.1.3.4) *c-char-sequence:*

    c-char
    c-char-sequence c-char

(3.1.3.4) *c-char:*

    any member of the source character set except
        the single-quote `'` , backslash `\`, or new-line character
    escape-sequence

(3.1.3.4) *escape-sequence:*

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

(3.1.3.4) *simple-escape-sequence:* one of

    `\'   \"   \?   \\`
    `\a  \b  \f  \n  \r  \t  \v`

(3.1.3.4) *octal-escape-sequence:*

    `\` *octal-digit*
    `\` *octal-digit octal-digit*
    `\` *octal-digit octal-digit octal-digit*

(3.1.3.4) *hexadecimal-escape-sequence:*

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

A.1.5 String Literals

(3.1.4) *string-literal:*

    `"`*s-char-sequence*$_{opt}$`"`
    `L"`*s-char-sequence*$_{opt}$`"`

(3.1.4) *s-char-sequence:*

    s-char
    s-char-sequence s-char

(3.1.4) *s-char:*

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

A.1.6 Operators

(3.1.5) *operator:* one of

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

A.1.7 Punctuators

(3.1.6) *punctuator:* one of

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

A.1.8 Header Names

(3.1.7) *header-name:*

>*<h-char-sequence>*
>*"q-char-sequence"*

(3.1.7) *h-char-sequence:*

>*h-char*
>*h-char-sequence h-char*

(3.1.7) *h-char:*

>any member of the source character set except
>   the new-line character and **>**

(3.1.7) *q-char-sequence:*

>*q-char*
>*q-char-sequence q-char*

(3.1.7) *q-char:*

>any member of the source character set except
>   the new-line character and **"**

A.1.9 Preprocessing Numbers

(3.1.8) *pp-number:*

>*digit*
>*. digit*
>*pp-number digit*
>*pp-number nondigit*
>*pp-number* **e** *sign*
>*pp-number* **E** *sign*
>*pp-number .*

A.2 Phrase Structure Grammar

A.2.1 Expressions

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

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

(3.3.2) *argument-expression-list:*
       assignment-expression
       argument-expression-list **,** *assignment-expression*

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

(3.3.3) *unary-operator:* one of
       &   *   +   -   ~   !

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

(3.3.5) *multiplicative-expression:*
       cast-expression
       multiplicative-expression ***** *cast-expression*
       multiplicative-expression **/** *cast-expression*
       multiplicative-expression **%** *cast-expression*

(3.3.6) *additive-expression:*
       multiplicative-expression
       additive-expression **+** *multiplicative-expression*
       additive-expression **-** *multiplicative-expression*

(3.3.7) *shift-expression:*
       additive-expression
       shift-expression **<<** *additive-expression*
       shift-expression **>>** *additive-expression*

(3.3.8) *relational-expression:*

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

(3.3.9) *equality-expression:*

> *relational-expression*
> *equality-expression* **==** *relational-expression*
> *equality-expression* **!=** *relational-expression*

(3.3.10) *AND-expression:*

> *equality-expression*
> *AND-expression* **&** *equality-expression*

(3.3.11) *exclusive-OR-expression:*

> *AND-expression*
> *exclusive-OR-expression* **^** *AND-expression*

(3.3.12) *inclusive-OR-expression:*

> *exclusive-OR-expression*
> *inclusive-OR-expression* **|** *exclusive-OR-expression*

(3.3.13) *logical-AND-expression:*

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

(3.3.14) *logical-OR-expression:*

> *logical-AND-expression*
> *logical-OR-expression* **||** *logical-AND-expression*

(3.3.15) *conditional-expression:*

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

(3.3.16) *assignment-expression:*

> *conditional-expression*
> *unary-expression* *assignment-operator* *assignment-expression*

(3.3.16) *assignment-operator:* one of

> **=** ***=** **/=** **%=** **+=** **-=** **<<=** **>>=** **&=** **^=** **|=**

(3.3.17) *expression:*

> *assignment-expression*
> *expression* **,** *assignment-expression*

(3.4) *constant-expression:*

> *conditional-expression*

A.2.2 Declarations

(3.5) *declaration:*

> *declaration-specifiers* *init-declarator-list*$_{opt}$ **;**

(3.5) *declaration-specifiers:*

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

(3.5) *init-declarator-list:*
>> *init-declarator*
>> *init-declarator-list* , *init-declarator*

(3.5) *init-declarator:*
>> *declarator*
>> *declarator* = *initializer*

(3.5.1) *storage-class-specifier:*
>> **typedef**
>> **extern**
>> **static**
>> **auto**
>> **register**

(3.5.2) *type-specifier:*
>> **void**
>> **char**
>> **short**
>> **int**
>> **long**
>> **float**
>> **double**
>> **signed**
>> **unsigned**
>> *struct-or-union-specifier*
>> *enum-specifier*
>> *typedef-name*

(3.5.2.1) *struct-or-union-specifier:*
>> *struct-or-union* *identifier*$_{opt}$ { *struct-declaration-list* }
>> *struct-or-union* *identifier*

(3.5.2.1) *struct-or-union:*
>> **struct**
>> **union**

(3.5.2.1) *struct-declaration-list:*
>> *struct-declaration*
>> *struct-declaration-list* *struct-declaration*

(3.5.2.1) *struct-declaration:*
>> *specifier-qualifier-list* *struct-declarator-list* ;

(3.5.2.1) *specifier-qualifier-list:*
>> *type-specifier* *specifier-qualifier-list*$_{opt}$
>> *type-qualifier* *specifier-qualifier-list*$_{opt}$

(3.5.2.1) *struct-declarator-list:*
>> *struct-declarator*
>> *struct-declarator-list* , *struct-declarator*

(3.5.2.1) *struct-declarator:*
>> *declarator*
>> *declarator*$_{opt}$: *constant-expression*

(3.5.2.2) *enum-specifier:*
>> **enum** *identifier*$_{opt}$ { *enumerator-list* }
>> **enum** *identifier*

(3.5.2.2) *enumerator-list:*
> *enumerator*
> *enumerator-list , enumerator*

(3.5.2.2) *enumerator:*
> *enumeration-constant*
> *enumeration-constant = constant-expression*

(3.5.3) *type-qualifier:*
> **const**
> **volatile**

(3.5.4) *declarator:*
> *pointer*$_{opt}$ *direct-declarator*

(3.5.4) *direct-declarator:*
> *identifier*
> **(** *declarator* **)**
> *direct-declarator* **[** *constant-expression*$_{opt}$ **]**
> *direct-declarator* **(** *parameter-type-list* **)**
> *direct-declarator* **(** *identifier-list*$_{opt}$ **)**

(3.5.4) *pointer:*
> ***** *type-qualifier-list*$_{opt}$
> ***** *type-qualifier-list*$_{opt}$ *pointer*

(3.5.4) *type-qualifier-list:*
> *type-qualifier*
> *type-qualifier-list type-qualifier*

(3.5.4) *parameter-type-list:*
> *parameter-list*
> *parameter-list , ...*

(3.5.4) *parameter-list:*
> *parameter-declaration*
> *parameter-list , parameter-declaration*

(3.5.4) *parameter-declaration:*
> *declaration-specifiers declarator*
> *declaration-specifiers abstract-declarator*$_{opt}$

(3.5.4) *identifier-list:*
> *identifier*
> *identifier-list , identifier*

(3.5.5) *type-name:*
> *specifier-qualifier-list abstract-declarator*$_{opt}$

(3.5.5) *abstract-declarator:*
> *pointer*
> *pointer*$_{opt}$ *direct-abstract-declarator*

(3.5.5) *direct-abstract-declarator:*
> **(** *abstract-declarator* **)**
> *direct-abstract-declarator*$_{opt}$ **[** *constant-expression*$_{opt}$ **]**
> *direct-abstract-declarator*$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

(3.5.6) *typedef-name:*
> *identifier*

(3.5.7) *initializer:*

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

(3.5.7) *initializer-list:*

> *initializer*
> *initializer-list* , *initializer*

A.2.3 Statements

(3.6) *statement:*

> *labeled-statement*
> *compound-statement*
> *expression-statement*
> *selection-statement*
> *iteration-statement*
> *jump-statement*

(3.6.1) *labeled-statement:*

> *identifier* : *statement*
> **case** *constant-expression* : *statement*
> **default** : *statement*

(3.6.2) *compound-statement:*

> { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

(3.6.2) *declaration-list:*

> *declaration*
> *declaration-list declaration*

(3.6.2) *statement-list:*

> *statement*
> *statement-list statement*

(3.6.3) *expression-statement:*

> *expression*$_{opt}$;

(3.6.4) *selection-statement:*

> **if** (*expression*) *statement*
> **if** (*expression*) *statement* **else** *statement*
> **switch** (*expression*) *statement*

(3.6.5) *iteration-statement:*

> **while** (*expression*) *statement*
> **do** *statement* **while** (*expression*) ;
> **for** (*expression*$_{opt}$; *expression*$_{opt}$; *expression*$_{opt}$) *statement*

(3.6.6) *jump-statement:*

> **goto** *identifier* ;
> **continue** ;
> **break** ;
> **return** *expression*$_{opt}$;

A.2.4 External Definitions

(3.7) *translation-unit:*
>>> *external-declaration*
>>> *translation-unit external-declaration*

(3.7) *external-declaration:*
>>> *function-definition*
>>> *declaration*

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

A.3 Preprocessing Directives

(3.8) *preprocessing-file:*
>>> *group$_{opt}$*

(3.8) *group:*
>>> *group-part*
>>> *group group-part*

(3.8) *group-part:*
>>> *pp-tokens$_{opt}$ new-line*
>>> *if-section*
>>> *control-line*

(3.8.1) *if-section:*
>>> *if-group elif-groups$_{opt}$ else-group$_{opt}$ endif-line*

(3.8.1) *if-group:*
>>> **# if**   *constant-expression new-line group$_{opt}$*
>>> **# ifdef**   *identifier new-line group$_{opt}$*
>>> **# ifndef** *identifier new-line group$_{opt}$*

(3.8.1) *elif-groups:*
>>> *elif-group*
>>> *elif-groups elif-group*

(3.8.1) *elif-group:*
>>> **# elif**   *constant-expression new-line group$_{opt}$*

(3.8.1) *else-group:*
>>> **# else**   *new-line group$_{opt}$*

(3.8.1) *endif-line:*
>>> **# endif**   *new-line*

control-line:

(3.8.2)	**# include**	*pp-tokens new-line*
(3.8.3)	**# define**	*identifier replacement-list new-line*
(3.8.3)	**# define**	*identifier lparen identifier-list$_{opt}$) replacement-list new-line*
(3.8.3)	**# undef**	*identifier new-line*
(3.8.4)	**# line**	*pp-tokens new-line*
(3.8.5)	**# error**	*pp-tokens$_{opt}$ new-line*
(3.8.6)	**# pragma**	*pp-tokens$_{opt}$ new-line*
(3.8.7)	**#**	*new-line*

(3.8.3) *lparen:*
>>> the left-parenthesis character without preceding white space

(3.8.3) *replacement-list:*

        pp-tokens$_{opt}$

(3.8) *pp-tokens:*

        preprocessing-token
        pp-tokens preprocessing-token

(3.8) *new-line:*

        the new-line character

B. Sequence Points

The following are the sequence points described in 2.1.2.3.

• The call to a function, after the arguments have been evaluated (3.3.2.2).

• The end of the first operand of the following operators: logical AND **&&** (3.3.13); logical OR **||** (3.3.14); conditional **?** (3.3.15); comma **,** (3.3.17).

• The end of a full expression: an initializer (3.5.7); the expression in an expression statement (3.6.3); the controlling expression of a selection statement (**if** or **switch**) (3.6.4); the controlling expression of a **while** or **do** statement (3.6.5); each of the three expressions of a **for** statement (3.6.5.3); the expression in a **return** statement (3.6.6.4).

C. Library Summary

C.1 Errors <errno.h>

```
EDOM
ERANGE
errno
```

C.2 Common Definitions <stddef.h>

```
NULL
offsetof(type,  member-designator)
ptrdiff_t
size_t
wchar_t
```

C.3 Diagnostics <assert.h>

```
NDEBUG
void assert(int expression);
```

C.4 Character Handling <ctype.h>

```
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

C.5 Localization <locale.h>

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
NULL
struct lconv
char *setlocale(int category, const char *locale);
struct lconv *localeconv(void);
```

C.6 Mathematics <math.h>

```
HUGE_VAL
double acos(double x);
double asin(double x);
double atan(double x);
double atan2(double y, double x);
double cos(double x);
double sin(double x);
double tan(double x);
double cosh(double x);
double sinh(double x);
double tanh(double x);
double exp(double x);
double frexp(double value, int *exp);
double ldexp(double x, int exp);
double log(double x);
double log10(double x);
double modf(double value, double *iptr);
double pow(double x, double y);
double sqrt(double x);
double ceil(double x);
double fabs(double x);
double floor(double x);
double fmod(double x, double y);
```

C.7 Nonlocal Jumps <setjmp.h>

```
jmp_buf
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

C.8 Signal Handling <signal.h>

```
sig_atomic_t
SIG_DFL
SIG_ERR
SIG_IGN
SIGABRT
SIGFPE
SIGILL
SIGINT
SIGSEGV
SIGTERM
void (*signal(int sig, void (*func)(int)))(int);
int raise(int sig);
```

C.9 Variable Arguments <stdarg.h>

```
va_list
void va_start(va_list ap, parmN);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

C.10 Input/Output <stdio.h>

```
_IOFBF
_IOLBF
_IONBF
BUFSIZ
EOF
FILE
FILENAME_MAX
FOPEN_MAX
fpos_t
L_tmpnam
NULL
SEEK_CUR
SEEK_END
SEEK_SET
size_t
stderr
stdin
stdout
TMP_MAX
int remove(const char *filename);
int rename(const char *old, const char *new);
FILE *tmpfile(void);
char *tmpnam(char *s);
int fclose(FILE *stream);
int fflush(FILE *stream);
FILE *fopen(const char *filename, const char *mode);
FILE *freopen(const char *filename, const char *mode,
        FILE *stream);
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int scanf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, va_list arg);
int fgetc(FILE *stream);
char *fgets(char *s, int n, FILE *stream);
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *s);
int putc(int c, FILE *stream);
```

```
int putchar(int c);
int puts(const char *s);
int ungetc(int c, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb,
     FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
     FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fseek(FILE *stream, long int offset, int whence);
int fsetpos(FILE *stream, const fpos_t *pos);
long int ftell(FILE *stream);
void rewind(FILE *stream);
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
```

C.11 General Utilities <stdlib.h>

```
EXIT_FAILURE
EXIT_SUCCESS
MB_CUR_MAX
NULL
RAND_MAX
div_t
ldiv_t
size_t
wchar_t
double atof(const char *nptr);
int atoi(const char *nptr);
long int atol(const char *nptr);
double strtod(const char *nptr, char **endptr);
long int strtol(const char *nptr, char **endptr, int base);
unsigned long int strtoul(const char *nptr, char **endptr,
        int base);
int rand(void);
void srand(unsigned int seed);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void abort(void);
int atexit(void (*func)(void));
void exit(int status);
char *getenv(const char *name);
int system(const char *string);
void *bsearch(const void *key, const void *base,
        size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
void qsort(void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));
int abs(int j);
div_t div(int numer, int denom);
long int labs(long int j);
ldiv_t ldiv(long int numer, long int denom);
int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

C.12 String Handling <string.h>

```
NULL
size_t
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char *s1, const char *s2, size_t n);
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
void *memset(void *s, int c, size_t n);
char *strerror(int errnum);
size_t strlen(const char *s);
```

C.13 Date and Time <time.h>

```
CLOCKS_PER_SEC
NULL
clock_t
time_t
size_t
struct tm
clock_t clock(void);
double difftime(time_t time1, time_t time0);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char *s, size_t maxsize,
        const char *format, const struct tm *timeptr);
```

D. Implementation Limits

The contents of a header **<limits.h>** are given below, in alphabetic order. The minimum magnitudes shown shall be replaced by implementation-defined magnitudes with the same sign. The values shall all be constant expressions suitable for use in **#if** preprocessing directives. The components are described further in 2.2.4.2.1.

```
#define CHAR_BIT                          8
#define CHAR_MAX        UCHAR_MAX or SCHAR_MAX
#define CHAR_MIN               0 or SCHAR_MIN
#define INT_MAX                      +32767
#define INT_MIN                      -32767
#define LONG_MAX                 +2147483647
#define LONG_MIN                 -2147483647
#define MB_LEN_MAX                        1
#define SCHAR_MAX                      +127
#define SCHAR_MIN                      -127
#define SHRT_MAX                     +32767
#define SHRT_MIN                     -32767
#define UCHAR_MAX                       255
#define UINT_MAX                      65535
#define ULONG_MAX                4294967295
#define USHRT_MAX                     65535
```

The contents of a header **<float.h>** are given below. The value of **FLT_RADIX** shall be a constant expression suitable for use in **#if** preprocessing directives. Values that need not be constant expressions shall be supplied for all other components. The components are described further in 2.2.4.2.2.

```
#define FLT_ROUNDS
```

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal or greater in magnitude (absolute value) to those shown, with the same sign:

```
#define DBL_DIG                          10
#define DBL_MANT_DIG
#define DBL_MAX_10_EXP                  +37
#define DBL_MAX_EXP
#define DBL_MIN_10_EXP                  -37
#define DBL_MIN_EXP
#define FLT_DIG                           6
#define FLT_MANT_DIG
#define FLT_MAX_10_EXP                  +37
#define FLT_MAX_EXP
#define FLT_MIN_10_EXP                  -37
#define FLT_MIN_EXP
#define FLT_RADIX                         2
#define LDBL_DIG                         10
#define LDBL_MANT_DIG
#define LDBL_MAX_10_EXP                 +37
#define LDBL_MAX_EXP
#define LDBL_MIN_10_EXP                 -37
#define LDBL_MIN_EXP
```

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal to or greater than those shown:

```
#define DBL_MAX                          1E+37
#define FLT_MAX                          1E+37
#define LDBL_MAX                         1E+37
```

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal to or less than those shown:

```
#define DBL_EPSILON                      1E-9
#define DBL_MIN                          1E-37
#define FLT_EPSILON                      1E-5
#define FLT_MIN                          1E-37
#define LDBL_EPSILON                     1E-9
#define LDBL_MIN                         1E-37
```

E. Common Warnings

An implementation may generate warnings in many situations, none of which is specified as part of the standard. The following are a few of the more common situations.

- A block with initialization of an object that has automatic storage duration is jumped into (3.1.2.4).

- An integer character constant includes more than one character or a wide character constant includes more than one multibyte character (3.1.3.4).

- The characters **/*** are found in a comment (3.1.7).

- An implicit narrowing conversion is encountered, such as the assignment of a **long int** or a **double** to an **int**, or a pointer to **void** to a pointer to any type other than a character type (3.2).

- An ''unordered'' binary operator (not comma, **&&** or **||**) contains a side-effect to an lvalue in one operand, and a side-effect to, or an access to the value of, the identical lvalue in the other operand (3.3).

- A function is called but no prototype has been supplied (3.3.2.2).

- The arguments in a function call do not agree in number and type with those of the parameters in a function definition that is not a prototype (3.3.2.2).

- An object is defined but not used (3.5).

- A value is given to an object of an enumeration type other than by assignment of an enumeration constant that is a member of that type, or an enumeration variable that has the same type, or the value of a function that returns the same enumeration type (3.5.2.2).

- An aggregate has a partly bracketed initialization (3.5.7).

- A statement cannot be reached (3.6).

- A statement with no apparent effect is encountered (3.6).

- A constant expression is used as the controlling expression of a selection statement (3.6.4).

- A function has **return** statements with and without expressions (3.6.6.4).

- An incorrectly formed preprocessing group is encountered while skipping a preprocessing group (3.8.1).

- An unrecognized **#pragma** directive is encountered (3.8.6).

F. Portability Issues

This appendix collects some information about portability that appears in the standard.

F.1 Unspecified Behavior

The following are unspecified:

- The manner and timing of static initialization (2.1.2).

- The behavior if a printable character is written when the active position is at the final position of a line (2.2.2).

- The behavior if a backspace character is written when the active position is at the initial position of a line (2.2.2).

- The behavior if a horizontal tab character is written when the active position is at or past the last defined horizontal tabulation position (2.2.2).

- The behavior if a vertical tab character is written when the active position is at or past the last defined vertical tabulation position (2.2.2).

- The representations of floating types (3.1.2.5).

- The order in which expressions are evaluated — in any order conforming to the precedence rules, even in the presence of parentheses (3.3).

- The order in which side effects take place (3.3).

- The order in which the function designator and the arguments in a function call are evaluated (3.3.2.2).

- The alignment of the addressable storage unit allocated to hold a bit-field (3.5.2.1).

- The layout of storage for parameters (3.7.1).

- The order in which **#** and **##** operations are evaluated during macro substitution (3.8.3.3).

- Whether **errno** is a macro or an external identifier (4.1.3).

- Whether **setjmp** is a macro or an external identifier (4.6.1.1).

- Whether **va_end** is a macro or an external identifier (4.8.1.3).

- 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 (4.9.7.11).

- The details of the value stored by the **fgetpos** function on success (4.9.9.1).

- The details of the value returned by the **ftell** function for a text stream on success (4.9.9.4).

- The order and contiguity of storage allocated by the **calloc**, **malloc**, and **realloc** functions (4.10.3).

- Which of two elements that compare as equal is returned by the **bsearch** function (4.10.5.1).

- The order in an array sorted by the **qsort** function of two elements that compare as equal (4.10.5.2).

- The encoding of the calendar time returned by the **time** function (4.12.2.3).

F.2 Undefined Behavior

The behavior in the following circumstances 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 (2.1.1.2).

- 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, a header name, or a comment (2.2.1).

- 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 (2.2.1.2).

- An unmatched ' or " character is encountered on a logical source line during tokenization (3.1).

- The same identifier is used more than once as a label in the same function (3.1.2.1).

- An identifier is used that is not visible in the current scope (3.1.2.1).

- Identifiers that are intended to denote the same entity differ in a character beyond the minimal significant characters (3.1.2).

- The same identifier has both internal and external linkage in the same translation unit (3.1.2.2).

- The value stored in a pointer that referred to an object with automatic storage duration is used (3.1.2.4).

- Two declarations of the same object or function specify types that are not compatible (3.1.2.6).

- An unspecified escape sequence is encountered in a character constant or a string literal (3.1.3.4).

- An attempt is made to modify a string literal of either form (3.1.4).

- A character string literal token is adjacent to a wide string literal token (3.1.4).

- 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 (3.1.7).

- An arithmetic conversion produces a result that cannot be represented in the space provided (3.2.1).

- An lvalue with an incomplete type is used in a context that requires the value of the designated object (3.2.2.1).

- The value of a void expression is used or an implicit conversion (except to **void**) is applied to a void expression (3.2.2.2).

- An object is modified more than once, or is modified and accessed other than to determine the new value, between two sequence points (3.3).

- 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) (3.3).

- 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 (3.3).

- An argument to a function is a void expression (3.3.2.2).

- For a function call without a function prototype, the number of arguments does not agree with the number of parameters (3.3.2.2).

- 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 (3.3.2.2).

- If a function is called with a function prototype and the function is not defined with a compatible type (3.3.2.2).

- A function that accepts a variable number of arguments is called without a function prototype that ends with an ellipsis (3.3.2.2).

- An invalid array reference, null pointer reference, or reference to an object declared with automatic storage duration in a terminated block occurs (3.3.3.2).

- 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 (3.3.4).

- 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 (3.3.4).

- A pointer is converted to other than an integral or pointer type (3.3.4).

- A pointer that does not behave like a pointer to an element of an array object is added to or subtracted from (3.3.6).

- Pointers that do not behave as if they point to the same array object are subtracted (3.3.6).

- 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 (3.3.7).

- Pointers are compared using a relational operator that do not point to the same aggregate or union (3.3.8).

- An object is assigned to an overlapping object (3.3.16.1).

- 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 (3.5).

- A function is declared at block scope with a storage-class specifier other than **extern** (3.5.1).

- A structure or union is defined as containing only unnamed members (3.5.2.1).

- A bit-field is declared with a type other than **int**, **signed int**, or **unsigned int** (3.5.2.1).

- An attempt is made to modify an object with const-qualified type by means of an lvalue with non-const-qualified type (3.5.3).

- An attempt is made to refer to an object with volatile-qualified type by means of an lvalue with non-volatile-qualified type (3.5.3).

- The value of an uninitialized object that has automatic storage duration is used before a value is assigned (3.5.7).

- 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 (3.5.7).

- The value of a function is used, but no value was returned (3.6.6.4).

- An identifier with external linkage is used but there does not exist exactly one external definition in the program for the identifier (3.7).

- A function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation (3.7.1).

- An identifier for an object with internal linkage and an incomplete type is declared with a tentative definition (3.7.2).

- The token **defined** is generated during the expansion of a **#if** or **#elif** preprocessing directive (3.8.1).

- The **#include** preprocessing directive that results after expansion does not match one of the two header name forms (3.8.2).

- A macro argument consists of no preprocessing tokens (3.8.3).

- There are sequences of preprocessing tokens within the list of macro arguments that would otherwise act as preprocessing directive lines (3.8.3).

- The result of the preprocessing operator **#** is not a valid character string literal (3.8.3.2).

- The result of the preprocessing concatenation operator **##** is not a valid preprocessing token (3.8.3.3).

- The **#line** preprocessing directive that results after expansion does not match one of the two well-defined forms (3.8.4).

- One of the following identifiers is the subject of a **#define** or **#undef** preprocessing directive: **defined**, **__LINE__**, **__FILE__**, **__DATE__**, **__TIME__**, or **__STDC__** (3.8.8).

- An attempt is made to copy an object to an overlapping object by use of a library function other than **memmove** (section 4).

- The effect if the program redefines a reserved external identifier (4.1.2).

- 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 (4.1.2).

- A macro definition of **errno** is suppressed to obtain access to an actual object (4.1.3).

- 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 (4.1.5).

- A library function argument has an invalid value, unless the behavior is specified explicitly (4.1.6).

- A library function that accepts a variable number of arguments is not declared (4.1.6).

- The macro definition of **assert** is suppressed to obtain access to an actual function (4.2).

- The argument to a character handling function is out of the domain (4.3).

- A macro definition of **setjmp** is suppressed to obtain access to an actual function (4.6).

- 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**) (4.6.1.1).

- 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 (4.6.2.1).

- The **longjmp** function is invoked from a nested signal routine (4.6.2.1).

- 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** (4.7.1.1).

- 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** (4.7.1.1).

- 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 (4.8).

- A macro definition of **va_start**, **va_arg**, or **va_end** or a combination thereof is suppressed to obtain access to an actual function (4.8.1).

- 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 (4.8.1.1).

- There is no actual next argument for a **va_arg** macro invocation (4.8.1.2).

- The type of the actual next argument in a variable argument list disagrees with the type specified by the **va_arg** macro (4.8.1.2).

- The **va_end** macro is invoked without a corresponding invocation of the **va_start** macro (4.8.1.3).

- A return occurs from a function with a variable argument list initialized by the **va_start** macro before the **va_end** macro is invoked (4.8.1.3).

- The stream for the **fflush** function points to an input stream or to an update stream in which the most recent operation was input (4.9.5.2).

- 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 (4.9.5.3).

- The format for the **fprintf** or **fscanf** function does not match the argument list (4.9.6).

- An invalid conversion specification is found in the format for the **fprintf** or **fscanf** function (4.9.6).

- A **%%** conversion specification for the **fprintf** or **fscanf** function contains characters between the pair of **%** characters (4.9.6).

- 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** (4.9.6.1).

- 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 (4.9.6.1).

- 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 (4.9.6.1).

- 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**) (4.9.6.1).

- A single conversion by the **fprintf** function produces more than 509 characters of output (4.9.6.1).

- 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** (4.9.6.2).

- 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 (4.9.6.2).

- 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 (4.9.6.2).

- The result of converting a string to a number by the **atof**, **atoi**, or **atol** function cannot be represented (4.10.1).

- The value of a pointer that refers to space deallocated by a call to the **free** or **realloc** function is referred to (4.10.3).

- 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** (4.10.3).

- A program executes more than one call to the **exit** function (4.10.4.3).

- The result of an integer arithmetic function (**abs**, **div**, **labs**, or **ldiv**) cannot be represented (4.10.6).

- 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 (4.10.7).

- An array written to by a copying or concatenation function is too small (4.11.2, 4.11.3).

- An invalid conversion specification is found in the format for the **strftime** function (4.12.3.5).

F.3 Implementation-Defined Behavior

Each implementation shall document its behavior in each of the areas listed in this section. The following are implementation-defined:

F.3.1 Translation

- How a diagnostic is identified (2.1.1.3).

F.3.2 Environment

- The semantics of the arguments to **main** (2.1.2.2.1).

- What constitutes an interactive device (2.1.2.3).

F.3.3 Identifiers

- The number of significant initial characters (beyond 31) in an identifier without external linkage (3.1.2).

- The number of significant initial characters (beyond 6) in an identifier with external linkage (3.1.2).

- Whether case distinctions are significant in an identifier with external linkage (3.1.2).

F.3.4 Characters

- The members of the source and execution character sets, except as explicitly specified in the standard (2.2.1).

- The shift states used for the encoding of multibyte characters (2.2.1.2).

- The number of bits in a character in the execution character set (2.2.4.2.1).

- The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (3.1.3.4).

- 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 (3.1.3.4).

- 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 (3.1.3.4).

- The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (3.1.3.4).

- Whether a "plain" **char** has the same range of values as **signed char** or **unsigned char** (3.2.1.1).

F.3.5 Integers

- The representations and sets of values of the various types of integers (3.1.2.5).

- 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 (3.2.1.2).

- The results of bitwise operations on signed integers (3.3).

- The sign of the remainder on integer division (3.3.5).

- The result of a right shift of a negative-valued signed integral type (3.3.7).

F.3.6 Floating Point

- The representations and sets of values of the various types of floating-point numbers (3.1.2.5).

- The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (3.2.1.3).

- The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (3.2.1.4).

F.3.7 Arrays and Pointers

- The type of integer required to hold the maximum size of an array — that is, the type of the **sizeof** operator, **size_t** (3.3.3.4, 4.1.1).

- The result of casting a pointer to an integer or vice versa (3.3.4).

- The type of integer required to hold the difference between two pointers to elements of the same array, **ptrdiff_t** (3.3.6, 4.1.1).

F.3.8 Registers

- The extent to which objects can actually be placed in registers by use of the **register** storage-class specifier (3.5.1).

F.3.9 Structures, Unions, Enumerations, and Bit-Fields

- A member of a union object is accessed using a member of a different type (3.3.2.3).

- The padding and alignment of members of structures (3.5.2.1). This should present no problem unless binary data written by one implementation are read by another.

- Whether a "plain" **int** bit-field is treated as a **signed int** bit-field or as an **unsigned int** bit-field (3.5.2.1).

- The order of allocation of bit-fields within a unit (3.5.2.1).

- Whether a bit-field can straddle a storage-unit boundary (3.5.2.1).

- The integer type chosen to represent the values of an enumeration type (3.5.2.2).

F.3.10 Qualifiers

- What constitutes an access to an object that has volatile-qualified type (3.5.5.3).

F.3.11 Declarators

- The maximum number of declarators that may modify an arithmetic, structure, or union type (3.5.4).

F.3.12 Statements

- The maximum number of **case** values in a **switch** statement (3.6.4.2).

F.3.13 Preprocessing Directives

- 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 (3.8.1).

- The method for locating includable source files (3.8.2).

- The support of quoted names for includable source files (3.8.2).

- The mapping of source file character sequences (3.8.2).

- The behavior on each recognized **#pragma** directive (3.8.6).

- The definitions for **__DATE__** and **__TIME__** when respectively, the date and time of translation are not available (3.8.8).

F.3.14 Library Functions

- The null pointer constant to which the macro **NULL** expands (4.1.5).

- The diagnostic printed by and the termination behavior of the **assert** function (4.2).

- The sets of characters tested for by the **isalnum**, **isalpha**, **iscntrl**, **islower**, **isprint**, and **isupper** functions (4.3.1).

- The values returned by the mathematics functions on domain errors (4.5.1).

- Whether the mathematics functions set the integer expression **errno** to the value of the macro **ERANGE** on underflow range errors (4.5.1).

- Whether a domain error occurs or zero is returned when the **fmod** function has a second argument of zero (4.5.6.4).

- The set of signals for the **signal** function (4.7.1.1).

- The semantics for each signal recognized by the **signal** function (4.7.1.1).

- The default handling and the handling at program startup for each signal recognized by the **signal** function (4.7.1.1).

- 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 (4.7.1.1).

- Whether the default handling is reset if the **SIGILL** signal is received by a handler specified to the **signal** function (4.7.1.1).

- Whether the last line of a text stream requires a terminating new-line character (4.9.2).

- Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (4.9.2).

- The number of null characters that may be appended to data written to a binary stream (4.9.2).

- Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file (4.9.3).

- Whether a write on a text stream causes the associated file to be truncated beyond that point (4.9.3).

- The characteristics of file buffering (4.9.3).

- Whether a zero-length file actually exists (4.9.3).

- The rules for composing valid file names (4.9.3).

- Whether the same file can be open multiple times (4.9.3).

- The effect of the **remove** function on an open file (4.9.4.1).

- The effect if a file with the new name exists prior to a call to the **rename** function (4.9.4.2).

- The output for **%p** conversion in the **fprintf** function (4.9.6.1).

- The input for **%p** conversion in the **fscanf** function (4.9.6.2).

- The interpretation of a − character that is neither the first nor the last character in the scanlist for **%[** conversion in the **fscanf** function (4.9.6.2).

- The value to which the macro **errno** is set by the **fgetpos** or **ftell** function on failure (4.9.9.1, 4.9.9.4).

- The messages generated by the **perror** function (4.9.10.4).

- The behavior of the **calloc**, **malloc**, or **realloc** function if the size requested is zero (4.10.3).

- The behavior of the **abort** function with regard to open and temporary files (4.10.4.1).

- The status returned by the **exit** function if the value of the argument is other than zero, **EXIT_SUCCESS**, or **EXIT_FAILURE** (4.10.4.3).

- The set of environment names and the method for altering the environment list used by the **getenv** function (4.10.4.4).

- The contents and mode of execution of the string by the **system** function (4.10.4.5).

- The contents of the error message strings returned by the **strerror** function (4.11.6.2).

- The local time zone and Daylight Saving Time (4.12.1).

- The era for the **clock** function (4.12.2.1).

F.4 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 (2.2.1).

- The direction of printing (2.2.2).

- The decimal-point character (4.1.1).

- The implementation-defined aspects of character testing and case mapping functions (4.3).

- The collation sequence of the execution character set (4.11.4.4).

- The formats for time and date (4.12.3.5).

F.5 Common Extensions

The following extensions are widely used in many systems, but are not portable to all implementations. The inclusion of any extension that may cause a strictly conforming program to become invalid renders an implementation nonconforming. Examples of such extensions are new keywords, or library functions declared in standard headers or predefined macros with names that do not begin with an underscore.

F.5.1 Environment Arguments

In a hosted environment, the **main** function receives a third argument, **char *envp[]**, that points to a null-terminated array of pointers to **char**, each of which points to a string that provides information about the environment for this execution of the process (2.1.2.2.1).

F.5.2 Specialized Identifiers

Characters other than the underscore _, letters, and digits, that are not defined in the required source character set (such as the dollar sign **$**, or characters in national character sets) may appear in an identifier (3.1.2).

F.5.3 Lengths and Cases of Identifiers

All characters in identifiers (with or without external linkage) are significant and case distinctions are observed (3.1.2).

F.5.4 Scopes of Identifiers

A function identifier, or the identifier of an object the declaration of which contains the keyword **extern**, has file scope (3.1.2.1).

F.5.5 Writable String Literals

String literals are modifiable. Identical string literals shall be distinct (3.1.4).

F.5.6 Other Arithmetic Types

Other arithmetic types, such as **long long int**, and their appropriate conversions are defined (3.2.2.1).

F.5.7 Function Pointer Casts

A pointer to an object or to **void** may be cast to a pointer to a function, allowing data to be invoked as a function (3.3.4). A pointer to a function may be cast to a pointer to an object or to **void**, allowing a function to be inspected or modified (for example, by a debugger) (3.3.4).

F.5.8 Non-int Bit-Field Types

Types other than **int**, **unsigned int**, or **signed int** can be declared as bit-fields, with appropriate maximum widths (3.5.2.1).

F.5.9 The **fortran** Keyword

The **fortran** declaration specifier may be used in a function declaration to indicate that calls suitable for FORTRAN should be generated, or that different representations for external names are to be generated (3.5.4.3).

F.5.10 The asm Keyword

The **asm** keyword may be used to insert assembly language code directly into the translator output. The most common implementation is via a statement of the form

> **asm** (*character-string-literal*) ;

(3.6).

F.5.11 Multiple External Definitions

There may be more than one external definition for the identifier of an object, with or without the explicit use of the keyword **extern**, If the definitions disagree, or more than one is initialized, the behavior is undefined (3.7.2).

F.5.12 Empty Macro Arguments

A macro argument may consist of no preprocessing tokens (3.8.3).

F.5.13 Predefined Macro Names

Macro names that do not begin with an underscore, describing the translation and execution environments, may be defined by the implementation before translation begins (3.8.8).

F.5.14 Extra Arguments for Signal Handlers

Handlers for specific signals may be called with extra arguments in addition to the signal number (4.7.1.1).

F.5.15 Additional Stream Types and File-Opening Modes

Additional mappings from files to streams may be supported (4.9.2), and additional file-opening modes may be specified by characters appended to the **mode** argument of the **fopen** function (4.9.5.3).

F.5.16 Defined File Position Indicator

The file position indicator is decremented by each successful call to the **ungetc** function for a text stream, except if its value was zero before a call (4.9.7.11).

Index

Only major references are listed.

Rationale for
American National Standard
for Information Systems –
Programming Language –
C

(This Rationale is not part of American National Standard X3.159-1989, but is included for information only.)

Contents

Section 1

INTRODUCTION

This Rationale summarizes the deliberations of X3J11, the Technical Committee charged by ANSI with devising a standard for the C programming language. It has been published along with the draft Standard to assist the process of formal public review.

The X3J11 Committee represents a cross-section of the C community: it consists of about fifty active members representing hardware manufacturers, vendors of compilers and other software development tools, software designers, consultants, academics, authors, applications programmers, and others. In the course of its deliberations, it has reviewed related American and international standards both published and in progress. It has attempted to be responsive to the concerns of the broader community: as of September 1988, it had received and reviewed almost 200 letters, including dozens of formal comments from the first public review, suggesting modifications and additions to the various preliminary drafts of the Standard.

Upon publication of the Standard, the primary role of the Committee will be to offer interpretations of the Standard. It will consider and respond to all correspondence received.

1.1 Purpose

The Committee's overall goal was to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments.

The X3J11 charter clearly mandates the Committee to *codify common existing practice*. The Committee has held fast to precedent wherever this was clear and unambiguous. The vast majority of the language defined by the Standard is precisely the same as is defined in Appendix A of *The C Programming Language* by Brian Kernighan and Dennis Ritchie, and as is implemented in almost all C translators. (This document is hereinafter referred to as K&R.)

K&R is not the only source of "existing practice." Much work has been done over

1

the years to improve the C language by addressing its weaknesses. The Committee has formalized enhancements of proven value which have become part of the various dialects of C.

Existing practice, however, has not always been consistent. Various dialects of C have approached problems in different and sometimes diametrically opposed ways. This divergence has happened for several reasons. First, K&R, which has served as the language specification for almost all C translators, is imprecise in some areas (thereby allowing divergent interpretations), and it does not address some issues (such as a complete specification of a library) important for code portability. Second, as the language has matured over the years, various extensions have been added in different dialects to address limitations and weaknesses of the language; these extensions have not been consistent across dialects.

One of the Committee's goals was to consider such areas of divergence and to establish a set of clear, unambiguous rules consistent with the rest of the language. This effort included the consideration of extensions made in various C dialects, the specification of a complete set of required library functions, and the development of a complete, correct syntax for C.

The work of the Committee was in large part a balancing act. The Committee has tried to improve portability while retaining the definition of certain features of C as machine-dependent. It attempted to incorporate valuable new ideas without disrupting the basic structure and fabric of the language. It tried to develop a clear and consistent language without invalidating existing programs. All of the goals were important and each decision was weighed in the light of sometimes contradictory requirements in an attempt to reach a workable compromise.

In specifying a standard language, the Committee used several guiding principles, the most important of which are:

Existing code is important, existing implementations are not. A large body of C code exists of considerable commercial value. Every attempt has been made to ensure that the bulk of this code will be acceptable to any implementation conforming to the Standard. The Committee did not want to force most programmers to modify their C programs just to have them accepted by a conforming translator.

On the other hand, no one implementation was held up as the exemplar by which to define C: it is assumed that all existing implementations must change somewhat to conform to the Standard.

C code can be portable. Although the C language was originally born with the UNIX operating system on the DEC PDP-11, it has since been implemented on a wide variety of computers and operating systems. It has also seen considerable use in cross-compilation of code for embedded systems to be executed in a free-standing environment. The Committee has attempted to specify the language and the library to be as widely implementable as possible, while recognizing that a system must meet certain minimum criteria to be considered a viable host or target for the language.

C code can be non-portable. Although it strove to give programmers the opportunity to write truly portable programs, the Committee did not want to *force*

programmers into writing portably, to preclude the use of C as a "high-level assembler": the ability to write machine-specific code is one of the strengths of C. It is this principle which largely motivates drawing the distinction between *strictly conforming program* and *conforming program* (§1.7).

Avoid "quiet changes." Any change to widespread practice altering the meaning of existing code causes problems. Changes that cause code to be so ill-formed as to require diagnostic messages are at least easy to detect. As much as seemed possible consistent with its other goals, the Committee has avoided changes that quietly alter one valid program to another with different semantics, that cause a working program to work differently without notice. In important places where this principle is violated, the Rationale points out a QUIET CHANGE.

A standard is a treaty between implementor and programmer. Some numerical limits have been added to the Standard to give both implementors and programmers a better understanding of what must be provided by an implementation, of what can be expected and depended upon to exist. These limits are presented as *minimum maxima* (i.e., lower limits placed on the values of upper limits specified by an implementation) with the understanding that any implementor is at liberty to provide higher limits than the Standard mandates. Any program that takes advantage of these more tolerant limits is not strictly conforming, however, since other implementations are at liberty to enforce the mandated limits.

Keep the spirit of C. The Committee kept as a major goal to preserve the traditional *spirit of C*. There are many facets of the spirit of C, but the essence is a community sentiment of the underlying principles upon which the C language is based. Some of the facets of the spirit of C can be summarized in phrases like

- *Trust the programmer.*
- *Don't prevent the programmer from doing what needs to be done.*
- *Keep the language small and simple.*
- *Provide only one way to do an operation.*
- *Make it fast, even if it is not guaranteed to be portable.*

The last proverb needs a little explanation. The potential for efficient code generation is one of the most important strengths of C. To help ensure that no code explosion occurs for what appears to be a very simple operation, many operations are defined to be *how the target machine's hardware does it* rather than by a general abstract rule. An example of this willingness to live with *what the machine does* can be seen in the rules that govern the widening of `char` objects for use in expressions: whether the values of `char` objects widen to signed or unsigned quantities typically depends on which byte operation is more efficient on the target machine.

One of the goals of the Committee was to avoid interfering with the ability of translators to generate compact, efficient code. In several cases the Committee has introduced features to improve the possible efficiency of the generated code; for instance, floating point operations may be performed in single-precision if both operands are `float` rather than `double`.

1.2   Scope

This Rationale focuses primarily on additions, clarifications, and changes made to the language as described in the Base Documents (see §1.5). It is *not* a rationale for the C language as a whole: the Committee was charged with codifying an existing language, not designing a new one. No attempt is made in this Rationale to defend the pre-existing syntax of the language, such as the syntax of declarations or the binding of operators.

The Standard is contrived as carefully as possible to permit a broad range of implementations, from direct interpreters to highly optimizing compilers with separate linkers, from ROM-based embedded microcomputers to multi-user multi-processing host systems. A certain amount of specialized terminology has therefore been chosen to minimize the bias toward compiler implementations shown in the Base Documents.

The Rationale discusses some language or library features which were *not* adopted into the Standard. These are usually features which are popular in some C implementations, so that a user of those implementations might question why they do not appear in the Standard.

1.3   References

1.4   Organization of the document

This Rationale is organized to parallel the Standard as closely as possible, to facilitate finding relevant discussions. Some subsections of the Rationale comprise just the subsection title from the Standard: this indicates that the Committee thought no special comment was necessary. Where a given discussion touches on several areas, attempts have been made to include cross-references within the text. Such references, unless they specify the Standard or the Rationale, are deliberately ambiguous.

As for the organization of the Standard itself, Base Documents existed only for Sections 3 (Language) and 4 (Library) of the Standard. Section 1 (Introduction) was modeled after the introductory matter in several other standards for procedural languages. Section 2 (Environment) was added to fill a need, identified from the start, to place a C program in context and describe the way it interacts with its surroundings. The Appendices were added as a repository for related material not included in the Standard itself, or to bring together in a single place information about a topic which was scattered throughout the Standard.

Just as the Standard proper excludes all examples, footnotes, references, and appendices, *this rationale is not part of the Standard.* The C language is defined by the Standard alone. If any part of this Rationale is not in accord with that definition, the Committee would very much like to be so informed.

1.5 Base documents

The Base Document for Section 3 (Language) was "The C Reference Manual" by Dennis M. Ritchie, which was used for several years within AT&T Bell Laboratories and reflects enhancements to C within the UNIX environment. A version of this manual was published as Appendix A of *The C Programming Language* by Kernighan and Ritchie (K&R). Several deviations in the Base Document from K&R were challenged during Committee deliberations, but most changes from K&R ultimately included in the Standard were readily endorsed by the Committee since they were widely known and accepted outside the UNIX user community.

The Base Document for Section 4 (Library) was the *1984 /usr/group Standard.* (/usr/group is a UNIX system users group.) In defining what a UNIX-like environment looks like to an applications programmer writing in C, /usr/group was obliged to describe library functions usable in *any* C environment. The Committee found /usr/group's work to be an excellent codification of existing practice in defining C libraries, once the UNIX-specific functions had been removed.

The work begun by /usr/group is being continued by the IEEE Committee 1003 to define a portable operating system interface ("POSIX") based on the UNIX environment. The X3J11 Committee has been working with IEEE 1003 to resolve potential areas of overlap or conflict between the two Committees. The result of this coordination has been to divide responsibility for standardizing library functions into two areas. Those functions needed for a C implementation in any environment are the responsibility of X3J11 and are included in the Standard. IEEE 1003 retains responsibility for those functions which are operating-system-specific; the (POSIX) standard will refer to the ANSI C Standard for C library function definitions.

Many of the discussions in this Rationale employ the formula *"feature X* has been changed (added, removed) because ... ." The changes (additions, removals) should be understood as being with respect to the appropriate Base Document.

1.6 Definitions of terms

The definitions of *object*, *bit*, *byte*, and *alignment* reflect a strong consensus, reached after considerable discussion, about the fundamental nature of the memory organization of a C environment:

- All objects in C must be representable as a contiguous sequence of bytes, each of which is at least 8 bits wide.

- A `char` (or `signed char` or `unsigned char`) occupies exactly one byte.

(Thus, for instance, on a machine with 36-bit *words*, a *byte* can be defined to consist of 9, 12, 18, or 36 bits, these numbers being all the exact divisors of 36 which are not less than 8.) These strictures codify the widespread presumption that any object can be treated as an array of characters, the size of which is given by the `sizeof` operator with that object's type as its operand.

These definitions do not preclude "holes" in `struct` objects. Such holes are in fact often mandated by alignment and packing requirements. The holes simply do not participate in representing the (composite) value of an object.

The definition of *object* does not employ the notion of type. Thus an object has no type in and of itself. However, since an object may only be designated by an *lvalue* (see §3.2.2.1), the phrase "the type of an object" is taken to mean, here and in the Standard, "the type of the lvalue designating this object," and "the value of an object" means "the contents of the object interpreted as a value of the type of the lvalue designating the object."

The concept of *multi-byte character* has been added to C to support very large character sets. See §2.2.1.2.

The terms *unspecified behavior*, *undefined behavior*, and *implementation-defined behavior* are used to categorize the result of writing programs whose properties the Standard does not, or cannot, completely describe. The goal of adopting this categorization is to allow a certain variety among implementations which permits *quality of implementation* to be an active force in the marketplace as well as to allow certain popular extensions, without removing the cachet of *conformance to the Standard*. Appendix F to the Standard catalogs those behaviors which fall into one of these three categories.

Unspecified behavior gives the implementor some latitude in translating programs. This latitude does not extend as far as failing to translate the program.

Undefined behavior gives the implementor license not to catch certain program errors that are difficult to diagnose. It also identifies areas of possible conforming language extension: the implementor may augment the language by providing a definition of the officially undefined behavior.

Implementation-defined behavior gives an implementor the freedom to choose the appropriate approach, but requires that this choice be explained to the user. Behaviors designated as implementation-defined are generally those in which a user could make meaningful coding decisions based on the implementation definition. Implementors should bear in mind this criterion when deciding how extensive an implementation definition ought to be. As with unspecified behavior, simply failing to translate the source containing the implementation-defined behavior is not an adequate response.

1.7   Compliance

The three-fold definition of compliance is used to broaden the population of conforming programs and distinguish between conforming programs using a single implementation and portable conforming programs.

A *strictly conforming program* is another term for a maximally portable program. The goal is to give the programmer a *fighting chance* to make powerful C programs that are also highly portable, without demeaning perfectly useful C programs that happen not to be portable. Thus the adverb *strictly*.

By defining conforming implementations in terms of the programs they accept, the Standard leaves open the door for a broad class of extensions as part of a conforming implementation. By defining both *conforming hosted* and *conforming freestanding* implementations, the Standard recognizes the use of C to write such programs as operating systems and ROM-based applications, as well as more conventional hosted applications. Beyond this two-level scheme, no additional subsetting is defined for C, since the Committee felt strongly that too many levels dilutes the effectiveness of a standard.

Conforming program is thus the most tolerant of all categories, since only one conforming implementation need accept a program to rule it conforming. The primary limitation on this license is §2.1.1.3.

Diverse sections of the Standard comprise the "treaty" between programmers and implementors regarding various name spaces — if the programmer follows the rules of the Standard the implementation will not impose any further restrictions or surprises:

- A strictly conforming program can use only a restricted subset of the identifiers that begin with underscore (§4.1.2). Identifiers and keywords are distinct (§3.1.1). Otherwise, programmers can use whatever internal names they wish; a conforming implementation is guaranteed not to use conflicting names of the form reserved to the programmer. (Note, however, the class of identifiers which are identified in §4.13 as possible future library names.)

- The external functions defined in, or called within, a portable program can be named whatever the programmer wishes, as long as these names are distinct from the external names defined by the Standard library (§4). External names in a maximally portable program must be distinct within the first 6 characters mapped into one case (§3.1.2).

- A maximally portable program cannot, of course, assume any language keywords other than those defined in the Standard.

- Each function called within a maximally portable program must either be defined within some source file of the program or else be a function in the Standard library.

One proposal long entertained by the Committee was to mandate that each implementation have a translate-time switch for turning off extensions and making a pure Standard-conforming implementation. It was pointed out, however, that virtually every translate-time switch setting effectively creates a different "implementation," however close may be the effect of translating with two different switch settings. Whether an implementor chooses to offer a family of conforming implementations, or to offer an assortment of non-conforming implementations along with one that conforms, was not the business of the Committee to mandate. The Standard therefore confines itself to describing conformance, and merely suggests areas where extensions will not compromise conformance.

Other proposals rejected more quickly were to provide a validation suite, and to provide the source code for an acceptable library. Both were recognized to be major undertakings, and both were seen to compromise the integrity of the Standard by giving concrete examples that might bear more weight than the Standard itself. The potential legal implications were also a concern.

Standardization of such tools as program consistency checkers and symbolic debuggers lies outside the mandate of the Committee. However, the Committee has taken pains to allow such programs to work with conforming programs and implementations.

1.8   Future directions

Section 2

ENVIRONMENT

Because C has seen widespread use as a cross-compiled language, a clear distinction must be made between translation and execution environments. The preprocessor, for instance, is permitted to evaluate the expression in a `#if` statement using the long integer arithmetic native to the translation environment: these integers must comprise at least 32 bits, but need not match the number of bits in the execution environment. Other translate-time arithmetic, however, such as type casting and floating arithmetic, must more closely model the execution environment regardless of translation environment.

2.1   Conceptual models

The *as if* principle is invoked repeatedly in this Rationale. The Committee has found that describing various aspects of the C language, library, and environment in terms of concrete models best serves discussion and presentation. Every attempt has been made to craft the models so that implementors are constrained only insofar as they must bring about the same result, *as if* they had implemented the presentation model; often enough the clearest model would make for the worst implementation.

2.1.1   Translation environment

2.1.1.1   Program structure

The terms *source file*, *external linkage*, *linked*, *libraries*, and *executable program* all imply a conventional compiler-linker combination. All of these concepts have shaped the semantics of C, however, and are inescapable even in an interpreted environment. Thus, while implementations are not required to support separate compilation and linking with libraries, in some ways they must behave *as if* they do.

2.1.1.2   Translation phases

Perhaps the greatest undesirable diversity among existing C implementations can be found in preprocessing. Admittedly a distinct and primitive language superimposed

9

upon C, the preprocessing commands accreted over time, with little central direction, and with even less precision in their documentation. This evolution has resulted in a variety of local features, each with its ardent adherents: the Base Document offers little clear basis for choosing one over the other.

The consensus of the Committee is that preprocessing should be simple and *overt*, that it should sacrifice power for clarity. For instance, the macro invocation `f(a, b)` should assuredly have two actual arguments, even if `b` expands to `c, d`; and the formal definition of `f` must call for exactly two arguments. Above all, the preprocessing sub-language should be specified precisely enough to minimize or eliminate dialect formation.

To clarify the nature of preprocessing, the translation from source text to tokens is spelled out as a number of separate phases. The separate phases need not actually be present in the translator, but the net effect must be *as if* they were. The phases need not be performed in a separate preprocessor, although the definition certainly permits this common practice. Since the preprocessor need not know anything about the specific properties of the target, a machine-independent implementation is permissible.

The Committee deemed that it was outside the scope of its mandate to require the output of the preprocessing phases be available as a separate translator output file.

The *phases of translation* are spelled out to resolve the numerous questions raised about the precedence of different parses. Can a `#define` begin a comment? (No.) Is backslash/new-line permitted within a trigraph? (No.) Must a comment be contained within one `#include` file? (Yes.) And so on. The Rationale section on preprocessing (§3.8) discusses the reasons for many of the particular decisions which shaped the specification of the phases of translation.

A backslash immediately before a new-line has long been used to continue string literals, as well as preprocessing command lines. In the interest of easing machine generation of C, and of transporting code to machines with restrictive physical line lengths, the Committee generalized this mechanism to permit *any* token to be continued by interposing a backslash/new-line sequence.

2.1.1.3 Diagnostics

By mandating some form of diagnostic message for any program containing a syntax error or constraint violation, the Standard performs two important services. First, it gives teeth to the concept of *erroneous program*, since a conforming implementation must distinguish such a program from a valid one. Second, it severely constrains the nature of extensions permissible to a conforming implementation.

The Standard says nothing about the nature of the diagnostic message, which could simply be "`syntax error`", with no hint of where the error occurs. (An implementation must, of course, describe what translator output constitutes a diagnostic message, so that the user can recognize it as such.) The Committee ulti-

mately decided that any diagnostic activity beyond this level is an issue of *quality of implementation*, and that market forces would encourage more useful diagnostics. Nevertheless, the Committee felt that at least some significant class of errors must be diagnosed, and the class specified should be recognizable by all translators.

The Standard does not forbid extensions, but such extensions must not invalidate strictly conforming programs. The translator must diagnose the use of such extensions, or allow them to be disabled as discussed in (Rationale) §1.7. Otherwise, extensions to a conforming C implementation lie in such realms as defining semantics for syntax to which no semantics is ascribed by the Standard, or giving meaning to *undefined behavior*.

2.1.2 Execution environments

The definition of *program startup* in the Standard is designed to permit initialization of static storage by executable code, as well as by data translated into the program image.

2.1.2.1 Freestanding environment

As little as possible is said about freestanding environments, since little is served by constraining them.

2.1.2.2 Hosted environment

The properties required of a hosted environment are spelled out in a fair amount of detail in order to give programmers a reasonable chance of writing programs which are portable among such environments.

The behavior of the arguments to `main`, and of the interaction of `exit`, `main` and `atexit` (see §4.10.4.2) has been codified to curb some unwanted variety in the representation of `argv` strings, and in the meaning of values returned by `main`.

The specification of `argc` and `argv` as arguments to `main` recognizes extensive prior practice. `argv[argc]` is required to be a null pointer to provide a redundant check for the end of the list, also on the basis of common practice.

`main` is the only function that may portably be declared either with zero or two arguments. (The number of arguments must ordinarily match exactly between invocation and definition.) This special case simply recognizes the widespread practice of leaving off the arguments to `main` when the program does not access the program argument strings. While many implementations support more than two arguments to `main`, such practice is neither blessed nor forbidden by the Standard; a program that defines `main` with three arguments is not *strictly conforming*. (See Standard Appendix F.5.1.)

Command line I/O redirection is not mandated by the Standard; this was deemed to be a feature of the underlying operating system rather than the C language.

2.1.2.3   Program execution

Because C expressions can contain side effects, issues of *sequencing* are important in expression evaluation. (See §3.3.) Most operators impose no sequencing requirements, but a few operators impose *sequence points* upon the evaluation: comma, logical-AND, logical-OR, and conditional. For example, in the expression (i = 1, a[i] = 0) the side effect (alteration to storage) specified by i = 1 must be completed before the expression a[i] = 0 is evaluated.

   Other sequence points are imposed by statement execution and completion of evaluation of a *full expression*. (See §3.6). Thus in fn(++a), the incrementation of a must be completed before fn is called. In i = 1; a[i] = 0; the side-effect of i = 1 must be complete before a[i] = 0 is evaluated.

   The notion of *agreement* has to do with the relationship between the *abstract machine* defining the semantics and an actual implementation. An *agreement point* for some object or class of objects is a sequence point at which the value of the object(s) in the real implementation must agree with the value prescribed by the abstract semantics.

   For example, compilers that hold variables in registers can sometimes drastically reduce execution times. In a loop like

```
sum = 0;
for (i = 0; i < N; ++i)
    sum += a[i];
```

both sum and i might be profitably kept in registers during the execution of the loop. Thus, the actual memory objects designated by sum and i would not change state during the loop.

   Such behavior is, of course, too loose for hardware-oriented applications such as device drivers and memory-mapped I/O. The following loop looks almost identical to the previous example, but the specification of volatile ensures that each assignment to *ttyport takes place in the same sequence, and with the same values, as the (hypothetical) abstract machine would have done.

```
volatile short *ttyport;
/* ... */
for (i = 0; i < N; ++i)
    *ttyport = a[i];
```

   Another common optimization is to pre-compute common subexpressions. In this loop:

```
volatile short *ttyport;
short mask1, mask2;
/* ... */
for (i = 0; i < N; ++i)
    *ttyport = a[i] & mask1 & mask2;
```

evaluation of the subexpression `mask1 & mask2` could be performed prior to the loop in the real implementation, assuming that neither `mask1` nor `mask2` appear as an operand of the address-of (`&`) operator anywhere in the function. In the abstract machine, of course, this subexpression is re-evaluated at each loop iteration, but the real implementation is not required to mimic this repetitiveness, because the variables `mask1` and `mask2` are not `volatile` and the same results are obtained either way.

The previous example shows that a subexpression can be pre-computed in the real implementation. A question sometimes asked regarding optimization is, "Is the rearrangement still conforming if the pre-computed expression might raise a signal (such as division by zero)?" Fortunately for optimizers, the answer is "Yes," because any evaluation that raises a computational signal has fallen into an *undefined behavior* (§3.3), for which any action is allowable.

Behavior is described in terms of an *abstract machine* to underscore, once again, that the Standard mandates results *as if* certain mechanisms are used, without requiring those actual mechanisms in the implementation. The Standard specifies agreement points at which the value of an object or class of objects in an implementation must agree with the value ascribed by the abstract semantics.

Appendix B to the Standard lists the sequence points specified in the body of the Standard.

The class of *interactive devices* is intended to include at least asynchronous terminals, or paired display screens and keyboards. An implementation may extend the definition to include other input and output devices, or even network inter-program connections, provided they obey the Standard's characterization of interactivity.

2.2 Environmental considerations

2.2.1 Character sets

The Committee ultimately came to remarkable unanimity on the subject of character set requirements. There was strong sentiment that C should not be tied to ASCII, despite its heritage and despite the precedent of Ada being defined in terms of ASCII. Rather, an implementation is required to provide a unique character code for each of the printable graphics used by C, and for each of the control codes representable by an escape sequence. (No particular graphic representation for any character is prescribed — thus the common Japanese practice of using the glyph ¥ for the C character `'\'` is perfectly legitimate.) Translation and execution environments may have different character sets, but each must meet this requirement in its own way. The goal is to ensure that a conforming implementation can translate a C translator written in C.

For this reason, and economy of description, source code is described *as if* it undergoes the same translation as text that is input by the standard library I/O routines: each line is terminated by some new-line character, regardless of its external representation.

2.2.1.1   Trigraph sequences

Trigraph sequences have been introduced as alternate spellings of some characters to allow the implementation of C in character sets which do not provide a sufficient number of non-alphabetic graphics.

Implementations are required to support these alternate spellings, even if the character set in use is ASCII, in order to allow transportation of code from systems which must use the trigraphs.

The Committee faced a serious problem in trying to define a character set for C. Not all of the character sets in general use have the right number of characters, nor do they support the graphical symbols that C users expect to see. For instance, many character sets for languages other than English resemble ASCII except that codes used for graphic characters in ASCII are instead used for extra alphabetic characters or diacritical marks. C relies upon a richer set of graphic characters than most other programming languages, so the representation of programs in character sets other than ASCII is a greater problem than for most other programming languages.

The International Standards Organization (ISO) uses three technical terms to describe character sets: *repertoire*, *collating sequence*, and *codeset*. The *repertoire* is the set of distinct printable characters. The term abstracts the notion of printable character from any particular representation; the glyphs R, $\mathcal{R}$, *R*, **R**, *R*, R, and $\Re$ all represent the same element of the repertoire, upper-case-R, which is distinct from lower-case-r. Having decided on the repertoire to be used (C needs a repertoire of 96 characters), one can then pick a *collating sequence* which corresponds to the internal representation in a computer. The repertoire and collating sequence together form the *codeset*.

What is needed for C is to determine the necessary repertoire, ignore the collating sequence altogether (it is of no importance to the language), and then find ways of expressing the repertoire in a way that should give no problems with currently popular codesets.

C derived its repertoire from the ASCII codeset. Unfortunately the ASCII repertoire is not a subset of all other commonly used character sets, and widespread practice in Europe is not to implement all of ASCII either, but use some parts of its collating sequence for special national characters.

The solution is an internationally agreed-upon repertoire, in terms of which an international representation of C can be defined. The ISO has defined such a standard: ISO 646 describes an *invariant subset* of ASCII.

The characters in the ASCII repertoire used by C and absent from the ISO 646 repertoire are:

```
    #  [  ]  {  }  \  |  ~  ^
```

Given this repertoire, the Committee faced the problem of defining representations for the absent characters. The obvious idea of defining two-character escape sequences fails because C uses all the characters which *are* in the ISO 646 repertoire:

no single escape character is available. The best that can be done is to use a *trigraph* — an escape digraph followed by a distinguishing character.

?? was selected as the escape digraph because it is not used anywhere else in C (except as noted below); it suggests that something unusual is going on. The third character was chosen with an eye to graphical similarity to the character being represented.

The sequence ?? cannot currently occur anywhere in a legal C program except in strings, character constants, comments, or header names. The character escape sequence '\?' (see §3.1.3.4) was introduced to allow two adjacent question-marks in such contexts to be represented as ?\?, a form distinct from the escape digraph.

The Committee makes no claims that a program written using trigraphs looks attractive. As a matter of style, it may be wise to surround trigraphs with white space, so that they stand out better in program text. Some users may wish to define preprocessing macros for some or all of the trigraph sequences.

QUIET CHANGE

Programs with character sequences such as ??! in string constants, character constants, or header names will now produce different results.

2.2.1.2   Multibyte characters

The "byte = character" orientation of C works well for text in Western alphabets, where the size of the character set is under 256. The fit is rather uncomfortable for languages such as Japanese and Chinese, where the repertoire of ideograms numbers in the thousands or tens of thousands.

Internally, such character sets can be represented as numeric codes, and it is merely necessary to choose the appropriate integral type to hold any such character.

Externally, whether in the files manipulated by a program, or in the text of the source files themselves, a conversion between these large codes and the various byte media is necessary.

The support in C of large character sets is based on these principles:

- Multibyte encodings of large character sets are necessary in I/O operations, in source text comments, and in source text string and character literals.

- No existing multibyte encoding is mandated in preference to any other; no widespread existing encoding should be precluded.

- The null character ('\0') may not be used as part of a multibyte encoding, except for the one-byte null character itself. This allows existing functions which manipulate strings transparently to work with multibyte sequences.

- Shift encodings (which interpret byte sequences in part on the basis of some state information) must start out in a known (default) shift state under certain circumstances, such as the start of string literals.

- The minimum number of absolutely necessary library functions is introduced. (See §4.10.7.)

2.2.2   Character display semantics

The Standard defines a number of internal character codes for specifying "format effecting actions on display devices," and provides printable escape sequences for each of them. These character codes are clearly modelled after ASCII control codes, and the mnemonic letters used to specify their escape sequences reflect this heritage. Nevertheless, they are *internal* codes for specifying the format of a display in an environment-independent manner; they must be written to a *text file* to effect formatting on a display device. The Standard states quite clearly that the external representation of a text file (or data stream) may well differ from the internal form, both in character codes and number of characters needed to represent a single internal code.

The distinction between internal and external codes most needs emphasis with respect to *new-line*. ANSI X3L2 (Codes and Character Sets) uses the term to refer to an external code used for information interchange whose display semantics specify a move to the next line. Both ANSI X3L2 and ISO 646 deprecate the combination of the motion to the next line with a motion to the initial position on the line. The C Standard, on the other hand, uses *new-line* to designate the end-of-line internal code represented by the escape sequence '\n'. While this ambiguity is perhaps unfortunate, use of the term in the latter sense is nearly universal within the C community. But the knowledge that this internal code has numerous external representations, depending upon operating system and medium, is equally widespread.

The alert sequence ('\a') has been added by popular demand, to replace, for instance, the ASCII BEL code explicitly coded as '\007'.

Proposals to add '\e' for ASCII ESC ('\033') were not adopted because other popular character sets such as EBCDIC have no obvious equivalent. (See §3.1.3.4.)

The vertical tab sequence ('\v') was added since many existing implementations support it, and since it is convenient to have a designation within the language for all the defined white space characters.

The semantics of the motion control escape sequences carefully avoid the Western language assumptions that printing advances left-to-right and top-to-bottom.

To avoid the issue of whether an implementation conforms if it cannot properly effect vertical tabs (for instance), the Standard emphasizes that the semantics merely describe *intent*.

2.2.3   Signals and interrupts

Signals are difficult to specify in a system-independent way. The Committee concluded that about the only thing a strictly conforming program can do in a signal handler is to assign a value to a `volatile static` variable which can be written

uninterruptedly and promptly return. (The header `<signal.h>` specifies a type `sig_atomic_t` which can be so written.) It is further guaranteed that a signal handler will not corrupt the automatic storage of an instantiation of any executing function, even if that function is called within the signal handler.

No such guarantees can be extended to library functions, with the explicit exceptions of `longjmp` (§4.6.2.1) and `signal` (§4.7.1.1), since the library functions may be arbitrarily interrelated and since some of them have profound effect on the environment.

Calls to `longjmp` are problematic, despite the assurances of §4.6.2.1. The signal could have occurred during the execution of some library function which was in the process of updating external state and/or static variables.

A second signal for the same handler could occur before the first is processed, and the Standard makes no guarantees as to what happens to the second signal.

2.2.4 Environmental limits

The Committee agreed that the Standard must say something about certain capacities and limitations, but just how to enforce these treaty points was the topic of considerable debate.

2.2.4.1 Translation limits

The Standard requires that an implementation be able to translate and compile some program that meets each of the stated limits. This criterion was felt to give a useful latitude to the implementor in meeting these limits. While a deficient implementation could probably contrive a program that meets this requirement, yet still succeed in being useless, the Committee felt that such ingenuity would probably require more work than making something useful. The sense of the Committee is that implementors should not construe the translation limits as the values of hard-wired parameters, but rather as a set of criteria by which an implementation will be judged.

Some of the limits chosen represent interesting compromises. The goal was to allow reasonably large portable programs to be written, without placing excessive burdens on reasonably small implementations.

The minimum maximum limit of 257 cases in a switch statement allows coding of lexical routines which can branch on any character (one of at least 256 values) or on the value EOF.

2.2.4.2 Numerical limits

In addition to the discussion below, see §4.1.4.

2.2.4.2.1 Sizes of integral types `<limits.h>`   Such a large body of C code has been developed for 8-bit byte machines that the integer sizes in such environments

must be considered normative. The prescribed limits are minima: an implementation on a machine with 9-bit bytes can be conforming, as can an implementation that defines `int` to be the same width as `long`. The negative limits have been chosen to accommodate ones-complement or sign-magnitude implementations, as well as the more usual twos-complement. The limits for the maxima and minima of unsigned types are specified as unsigned constants (e.g., `65535u`) to avoid surprising widenings of expressions involving these extrema.

The macro `CHAR_BIT` makes available the number of bits in a `char` object. The Committee saw little utility in adding such macros for other data types.

The names associated with the `short int` types (`SHRT_MIN`, etc., rather than `SHORT_MIN`, etc.) reflect prior art rather than obsessive abbreviation on the Committee's part.

2.2.4.2.2   Characteristics of floating types `<float.h>`

The characterization of floating point follows, with minor changes, that of the FORTRAN standardization committee (X3J3).[1] The Committee chose to follow the FORTRAN model in some part out of a concern for FORTRAN-to-C translation, and in large part out of deference to the FORTRAN committee's greater experience with fine points of floating point usage. Note that the floating point model adopted permits all common representations, including sign-magnitude and twos-complement, but precludes a logarithmic implementation.

Single precision (32-bit) floating point is considered adequate to support a conforming C implementation. Thus the minimum maxima constraining floating types are extremely permissive.

The Committee has also endeavored to accommodate the IEEE 754 floating point standard by not adopting any constraints on floating point which are contrary to this standard.

The term `FLT_MANT_DIG` stands for "float mantissa digits." The Standard now uses the more precise term *significand* rather than *mantissa*.

---

[1]See X3J3 working document S8-112.

Section 3

LANGUAGE

While more formal methods of language definition were explored, the Committee decided early on to employ the style of the Base Document: Backus-Naur Form for the syntax and prose for the constraints and semantics. Anything more ambitious was considered to be likely to delay the Standard, and to make it less accessible to its audience.

3.1 Lexical Elements

The Standard endeavors to bring preprocessing more closely into line with the token orientation of the language proper. To do so requires that at least some information about white space be retained through the early phases of translation (see §2.1.1.2). It also requires that an inverse mapping be defined from tokens back to source characters (see §3.8.3).

3.1.1 Keywords

Several keywords have been added: `const`, `enum`, `signed`, `void`, and `volatile`.

As much as possible, however, new features have been added by overloading existing keywords, as, for example, `long double` instead of `extended`. It is recognized that each added keyword will require some existing code that used it as an identifier to be rewritten. No meaningful programs are known to be quietly changed by adding the new keywords.

The keywords `entry`, `fortran`, and `asm` have not been included since they were either never used, or are not portable. Uses of `fortran` and `asm` as keywords are noted as *common extensions*.

3.1.2 Identifiers

While an implementation is not obliged to remember more than the first 31 characters of an identifier for the purpose of name matching, the programmer is effectively prohibited from intentionally creating two different identifiers that are the same in

the first 31 characters. Implementations may therefore store the full identifier; they are not obliged to truncate to 31.

The decision to extend significance to 31 characters for internal names was made with little opposition, but the decision to retain the old six-character case-insensitive restriction on significance of external names was most painful. While strong sentiment was expressed for making C "right" by requiring longer names everywhere, the Committee recognized that the language must, for years to come, coexist with other languages and with older assemblers and linkers. Rather than undermine support for the Standard, the severe restrictions have been retained.

The Committee has decided to label as *obsolescent* the practice of providing different identifier significance for internal and external identifers, thereby signalling its intent that some future version of the C Standard require 31-character case-sensitive external name significance, and thereby encouraging new implementations to support such significance.

Three solutions to the external identifier length/case problem were explored, each with its own set of problems:

1. *Label any C implementation without at least 31-character, case-sensitive significance in external identifiers as non-standard.*   This is unacceptable since the whole reason for a standard is portability, and many systems today simply do not provide such a name space.

2. *Require a C implementation which cannot provide 31-character, case-sensitive significance to map long identifiers into the identifier name space that it can provide.*   This option quickly becomes very complex for large, multi-source programs, since a program-wide database has to be maintained for all modules to avoid giving two different identifiers the same actual external name. It also reduces the usefulness of source code debuggers and cross reference programs, which generally work with the short mapped names, since the source-code name used by the programmer would likely bear little resemblance to the name actually generated.

3. *Require a C implementation which cannot provide 31-character, case-sensitive significance to rewrite the linker, assembler, debugger, any other language translators which use the linker, etc.*   This is not always practical, since the C implementor might not be providing the linker, etc. Indeed, on some systems only the manufacturer's linker can be used, either because the format of the resulting program file is not documented, or because the ability to create program files is restricted to secure programs.

Because of the decision to restrict significance of external identifiers to six case-insensitive characters, C programmers are faced with these choices when writing portable programs:

1. Make sure that external identifiers are unique within the first six characters,

and use only one case within the name. A unique six-character prefix could be used, followed by an underscore, followed by a longer, more descriptive name:

```
extern int a_xvz_real_long_name;
extern int a_rwt_real_long_name2;
```

2. Use the prefix method described above, and then use #define statements to provide a longer, more descriptive name for the unique name, such as:

```
#define real_long_name  a_xvz_real_long_name
#define real_long_name2 a_rwt_real_long_name2
```

Note that overuse of this technique might result in exceeding the limit on the number of allowed #define macros, or some other implementation limit.

3. Use longer and/or multi-case external names, and limit the portability of the programs to systems that support the longer names.

4. Declare all exported items (or pointers thereto) in a single data structure and export that structure. The technique can reduce the number of external identifiers to one per translation unit; member names within the structure are internal identifiers, hence can have full significance. The principal drawback of this technique is that functions can only be exported by reference, not by name; on many systems this entails a run-time overhead on each function call.

QUIET CHANGE

A program that depends upon internal identifiers matching only in the first (say) eight characters may change to one with distinct objects for each variant spelling of the identifier.

3.1.2.1 Scopes of identifiers

The Standard has separated from the overloaded keywords for storage classes the various concepts of *scope*, *linkage*, *name space*, and *storage duration*. (See §3.1.2.2, §3.1.2.3, §3.1.2.4.) This has traditionally been a major area of confusion.

One source of dispute was whether identifiers with external linkage should have file scope even when introduced within a block. The Base Document is vague on this point, and has been interpreted differently by different implementations. For example, the following fragment would be valid in the file scope scheme, while invalid in the block scope scheme:

```
typedef struct data d_struct ;

first(){
        extern d_struct func();
        /* ... */
}
```

```
second(){
        d_struct n = func();
}
```

While it was generally agreed that it is poor practice to take advantage of an external declaration once it had gone out of scope, some argued that a translator had to remember the declaration for checking anyway, so why not acknowledge this? The compromise adopted was to decree essentially that block scope rules apply, but that a conforming implementation need not diagnose a failure to redeclare an external identifier that had gone out of scope (*undefined behavior*).

QUIET CHANGE

A program relying on file scope rules may be valid under block scope rules but behave differently — for instance, if d_struct were defined as type float rather than struct data in the example above.

Although the scope of an identifier in a function prototype begins at its declaration and ends at the end of that function's declarator, this scope is of course ignored by the preprocessor. Thus an identifier in a prototype having the same name as that of an existing macro is treated as an invocation of that macro. For example:

```
#define status 23
void exit(int status);
```

generates an error, since the prototype after preprocessing becomes

```
void exit(int 23);
```

Perhaps more surprising is what happens if status is defined

```
#define status []
```

Then the resulting prototype is

```
void exit(int []);
```

which is syntactically correct but semantically quite different from the intent.

To protect an implementation's header prototypes from such misinterpretation, the implementor must write them to avoid these surprises. Possible solutions include not using identifiers in prototypes, or using names (such as __status or _Status) in the reserved name space.

3.1.2.2 Linkages of identifiers

The Standard requires that the first declaration, implicit or explicit, of an identifier specify (by the presence or absence of the keyword `static`) whether the identifier has internal or external linkage. This requirement allows for one-pass compilation in an implementation which must treat internal linkage items differently than external linkage items. An example of such an implementation is one which produces intermediate assembler code, and which therefore must construct names for internal linkage items to circumvent identifier length and/or case restrictions in the target assembler.

Existing practice in this area is inconsistent. Some implementations have avoided the renaming problem simply by restricting internal linkage names by the same rules as for external linkage. Others have disallowed a static declaration followed later by a defining instance, even though such constructs are necessary to declare mutually recursive static functions. The requirements adopted in the Standard may call for changes in some existing programs, but allow for maximum flexibility.

The definition model to be used for objects with external linkage was a major standardization issue. The basic problem was to decide which declarations of an object define storage for the object, and which merely reference an existing object. A related problem was whether multiple definitions of storage are allowed, or only one is acceptable. Existing implementations of C exhibit at least four different models, listed here in order of increasing restrictiveness:

Common Every object declaration with external linkage (whether or not the keyword `extern` appears in the declaration) creates a definition of storage. When all of the modules are combined together, each definition with the same name is located at the same address in memory. (The name is derived from *common storage* in FORTRAN.) This model was the intent of the original designer of C, Dennis Ritchie.

Relaxed Ref/Def The appearance of the keyword `extern` (whether it is used outside of the scope of a function or not) in a declaration indicates a pure reference (ref), which does not define storage. Somewhere in all of the translation units, at least one definition (def) of the object must exist. An external definition is indicated by an object declaration in file scope containing no storage class indication. A reference without a corresponding definition is an error. Some implementations also will not generate a reference for items which are declared with the `extern` keyword, but are never used within the code. The UNIX operating system C compiler and linker implement this model, which is recognized as a *common extension* to the C language (F.4.11). UNIX C programs which take advantage of this model are standard conforming in their environment, but are not maximally portable.

Strict Ref/Def This is the same as the relaxed ref/def model, save that only one definition is allowed. Again, some implementations may decide not to put out

references to items that are not used. This is the model specified in K&R and in the Base Document.

Initialization This model requires an explicit initialization to define storage. All other declarations are references.

Figure 3.1 demonstrates the differences between the models.

The model adopted in the Standard is a combination of features of the strict ref/def model and the initialization model. As in the strict ref/def model, only a single translation unit contains the definition of a given object — many environments cannot effectively or efficiently support the "distributed definition" inherent in the common or relaxed ref/def approaches. However, either an initialization, or an appropriate declaration without storage class specifier (see §3.7), serves as the external definition. This composite approach was chosen to accommodate as wide a range of environments and existing implementations as possible.

3.1.2.3   Name spaces of identifiers

Implementations have varied considerably in the number of separate name spaces maintained. The position adopted in the Standard is to permit as many separate name spaces as can be distinguished by context, except that all tags (`struct`, `union`, and `enum`) comprise a single name space.

3.1.2.4   Storage durations of objects

It was necessary to clarify the effect on automatic storage of jumping into a block that declares local storage. (See §3.6.2.) While many implementations allocate the maximum depth of automatic storage upon entry to a function, some explicitly allocate and deallocate on block entry and exit. The latter are required to assure that local storage is allocated regardless of the path into the block (although initializers in automatic declarations are not executed unless the block is entered from the top).

To effect true reentrancy for functions in the presence of signals raised asynchronously (see §2.2.3), an implementation must assure that the storage for function return values has automatic duration. This means that the caller must allocate automatic storage for the return value and communicate its location to the called function. (The typical case of return registers for small types conforms to this requirement: the calling convention of the implementation implicitly communicates the return location to the called function.)

3.1.2.5   Types

Several new types have been added:

```
void
void *
signed char
```

Figure 3.1: Comparison of identifier linkage models

Model	File 1	File 2
common	```extern int i;
main() {
i = 1;
second();
}```	```extern int i;
second() {
third(i);
}```
Relaxed Ref/Def	```int i;
main() {
i = 1;
second();
}```	```int i;
second() {
third(i);
}```
Strict Ref/Def	```int i;
main() {
i = 1;
second();
}```	```extern int i;
second() {
third(i);
}```
Initializer	```int i = 0;
main() {
i = 1;
second();
}```	```int i;
second() {
third(i);
}```

```
unsigned char
unsigned short
unsigned long
long double
```

New designations for existing types have been added:

```
signed short for short
signed int   for int
signed long  for long
```

void is used primarily as the typemark for a function which returns no result. It may also be used, in any context where the value of an expression is to be discarded, to indicate explicitly that a value is ignored by writing the cast (void). Finally, a function prototype list that has no arguments is written as f(void), because f() retains its old meaning that nothing is said about the arguments.

A "pointer to void," void *, is a generic pointer, capable of pointing to any (data) object without truncation. A pointer to void must have the same representation and alignment as a pointer to character; the intent of this rule is to allow existing programs which call library functions (such as memcpy and free) to continue to work. A pointer to void may not be dereferenced, although such a pointer may be converted to a normal pointer type which may be dereferenced. Pointers to other types coerce silently to and from void * in assignments, function prototypes, comparisons, and conditional expressions, whereas other pointer type clashes are invalid. It is undefined what will happen if a pointer of some type is converted to void *, and then the void * pointer is converted to a type with a stricter alignment requirement.

Three types of char are specified: signed, plain, and unsigned. A plain char may be represented as either signed or unsigned, depending upon the implementation, as in prior practice. The type signed char was introduced to make available a one-byte signed integer type on those systems which implement plain char as unsigned. For reasons of symmetry, the keyword signed is allowed as part of the type name of other integral types.

Two varieties of the integral types are specified: signed and unsigned. If neither specifier is used, signed is assumed. In the Base Document the only unsigned type is unsigned int.

The keyword unsigned is something of a misnomer, suggesting as it does arithmetic that is non-negative but capable of overflow. The semantics of the C type unsigned is that of modulus, or wrap-around, arithmetic, for which overflow has no meaning. The result of an unsigned arithmetic operation is thus always defined, whereas the result of a signed operation may (in principle) be undefined. In practice, on twos-complement machines, both types often give the same result for all operators except division, modulus, right shift, and comparisons. Hence there has been a lack of sensitivity in the C community to the differences between signed and unsigned arithmetic (see §3.2.1.1).

The Committee has explicitly restricted the C language to binary architectures, on the grounds that this stricture was implicit in any case:

- Bit-fields are specified by a number of bits, with no mention of "invalid integer" representation. The only reasonable encoding for such bit-fields is binary.

- The integer formats for `printf` suggest no provision for "illegal integer" values, implying that any result of bitwise manipulation produces an integer result which can be printed by `printf`.

- All methods of specifying integer constants — decimal, hex, and octal — specify an integer value. No method independent of integers is defined for specifying "bit-string constants." Only a binary encoding provides a complete one-to-one mapping between bit strings and integer values.

The restriction to "binary numeration systems" rules out such curiosities as Gray code, and makes possible arithmetic definitions of the bitwise operators on unsigned types (see §3.3.3.3, §3.3.7, §3.3.10, §3.3.11, §3.3.12).

A new floating type `long double` has been added to C. The `long double` type must offer at least as much precision as the type `double`. Several architectures support more than two floating types and thus can map a distinct machine type onto this additional C type. Several architectures which only support two floating point types can also take advantage of the three C types by mapping the less precise type onto `float` and `double`, and designating the more precise type `long double`. Architectures in which this mapping might be desirable include those in which single-precision floats offer at least as much precision as most other machines's double-precision, or those on which single-precision is considerably more efficient than double-precision. Thus the common C floating types would map onto an efficient implementation type, but the more precise type would still be available to those programmers who require its use.

To avoid confusion, `long float` as a synonym for `double` has been retired.

Enumerations permit the declaration of named constants in a more convenient and structured fashion than `#define`'s. Both enumeration constants and variables behave like integer types for the sake of type checking, however.

The Committee considered several alternatives for enumeration types in C:

1. leave them out;

2. include them as definitions of integer constants;

3. include them in the weakly typed form of the UNIX C compiler;

4. include them with strong typing, as, for example, in Pascal.

The Committee adopted the second alternative on the grounds that this approach most clearly reflects common practice. Doing away with enumerations altogether would invalidate a fair amount of existing code; stronger typing than integer creates problems, for instance, with arrays indexed by enumerations.

3.1.2.6   Compatible type and composite type

The notions of *compatible types* and *composite type* have been introduced to discuss those situations in which type declarations need not be identical. These terms are especially useful in explaining the relationship between an incomplete type and a complete type.

Structure, union, or enumeration type declarations in two different translation units do not formally declare the *same type*, even if the text of these declarations come from the same include file, since the translation units are themselves disjoint. The Standard thus specifies additional compatibility rules for such types, so that if two such declarations are sufficiently similar they are compatible.

3.1.3   Constants

In folding and converting constants, an implementation must use at least as much precision as is provided by the target environment. However, it is not required to use exactly the same precision as the target, since this would require a cross compiler to simulate target arithmetic at translation time.

The Committee considered the introduction of structure constants. Although it agreed that structure literals would occasionally be useful, its policy has been not to invent new features unless a strong need exists. Since the language already allows for initialized `const` structure objects, the need for inline anonymous structured constants seems less than pressing.

Several implementation difficulties beset structure constants. All other forms of constants are "self typing" — the type of the constant is evident from its lexical structure. Structure constants would require either an explicit type mark, or typing by context; either approach is considered to require increased complexity in the design of the translator, and either approach would also require as much, if not more, care on the part of the programmer as using an initialized structure object.

3.1.3.1   Floating constants

Consistent with existing practice, a floating point constant has been defined to have type `double`. Since the Standard now allows expressions that contain only `float` operands to be performed in `float` arithmetic (see §3.2.1.5) rather than `double`, a method of expressing explicit `float` constants is desirable. The new `long double` type raises similar issues.

Thus the `F` and `L` suffixes have been added to convey type information with floating constants, much like the `L` suffix for long integers. The default type of floating constants remains `double`, for compatibility with prior practice. Lower case `f` and `l` are also allowed as suffixes.

Note that the run-time selection of the decimal point character by `setlocale` (§4.4.1) has no effect on the syntax of C source text: the decimal point character is always period.

3.1.3.2 Integer constants

The rule that the default type of a decimal integer constant is either `int`, `long`, or `unsigned long`, depending on which type is large enough to hold the value without overflow, simplifies the use of constants.

The suffixes `U` and `u` have been added to specify unsigned numbers.

Unlike decimal constants, octal and hexadecimal constants too large to be `ints` are typed as `unsigned int` (if within range of that type), since it is more likely that they represent bit patterns or masks, which are generally best treated as unsigned, rather than "real" numbers.

Little support was expressed for the old practice of permitting the digits 8 and 9 in an octal constant, so it has been dropped.

A proposal to add binary constants was rejected due to lack of precedent and insufficient utility.

Despite a concern that a lower-case L could be taken for the numeral one at the end of an integral (or floating) literal, the Committee rejected proposals to remove this usage, primarily on the grounds of sanctioning existing practice.

The rules given for typing integer constants were carefully worked out in accordance with the Committee's deliberations on integral promotion rules (see §3.2.1.1).

QUIET CHANGE

Unsuffixed integer constants may have different types. In K&R, unsuffixed decimal constants greater than `INT_MAX`, and unsuffixed octal or hexadecimal constants greater than `UINT_MAX` are of type `long`.

3.1.3.3 Enumeration constants

Whereas an enumeration variable may have any integer type that correctly represents all its values when widened to `int`, an enumeration constant is only usable as the value of an expression. Hence its type is simply `int`. (See §3.1.2.5.)

3.1.3.4 Character constants

The digits 8 and 9 are no longer permitted in octal escape sequences. (Cf. octal constants, §3.1.3.2.)

The alert escape sequence has been added (see §2.2.2).

Hexadecimal escape sequences, beginning with `\x`, have been adopted, with precedent in several existing implementations. (Little sentiment was garnered for providing `\X` as well.) The escape sequence extends to the first non-hex-digit character, thus providing the capability of expressing any character constant no matter how large the type `char` is. String concatenation can be used to specify a hex-digit character following a hexadecimal escape sequence:

```
char a[] = "\xff" "f" ;
char b[] = {'\xff', 'f', '\0'};
```

These two initializations give a and b the same string value.

The Committee has chosen to reserve all lower case letters not currently used for future escape sequences (*undefined behavior*). All other characters with no current meaning are left to the implementor for extensions (*implementation-defined behavior*). No portable meaning is assigned to multi-character constants or ones containing other than the mandated source character set (*implementation-defined behavior*).

The Committee considered proposals to add the character constant '\e' to represent the ASCII ESC ('\033') character. This proposal was based upon the use of ESC as the initial character of most control sequences in common terminal driving disciplines, such as ANSI X3.64. However, this usage has no obvious counterpart in other popular character codes, such as EBCDIC. A programmer merely wishing to avoid having to type \033 to represent the ESC character in an ASCII/X3.64 environment, may, instead of writing

```
printf("\033[10;10h%d\n", somevalue);
```

write:

```
#define  ESC  "\033"

printf( ESC "[10;10h%d\n", somevalue);
```

Notwithstanding the general rule that literal constants are non-negative[1], a character constant containing one character is effectively preceded with a (char) cast and hence may yield a negative value if plain char is represented the same as signed char. This simply reflects widespread past practice and was deemed too dangerous to change.

QUIET CHANGE

A constant of the form '\078' is valid, but now has different meaning. It now denotes a character constant whose value is the (implementation-defined) combination of the values of the two characters '\07' and '8'. In some implementations the old meaning is the character whose code is $078 \equiv 0100 \equiv 64$.

QUIET CHANGE

A constant of the form '\a' or '\x' now may have different meaning. The old meaning, if any, was implementation dependent.

An L prefix distinguishes wide character constants. (See §2.2.1.2.)

---

[1] -3 is an expression: unary minus with operand 3.

3.1.4   String literals

String literals are specified to be unmodifiable. This specification allows implementations to share copies of strings with identical text, to place string literals in read-only memory, and perform certain optimizations. However, string literals do not have the type *array of const char*, in order to avoid the problems of pointer type checking, particularly with library functions, since assigning a *pointer to const char* to a plain *pointer to char* is not valid. Those members of the Committee who insisted that string literals should be modifiable were content to have this practice designated a common extension (see F.5.5).

Existing code which modifies string literals can be made strictly conforming by replacing the string literal with an initialized static character array. For instance,

```
char *p, *make_temp(char *str);
    /* ... */
p = make_temp("tempXXX");
        /* make_temp overwrites the literal */
        /* with a unique name */
```

can be changed to:

```
char *p, *make_temp(char *str);
    /* ... */
{
    static char template[ ] = "tempXXX";
    p = make_temp( template );
}
```

A long string can be continued across multiple lines by using the backslash-newline line continuation, but this practice requires that the continuation of the string start in the first position of the next line. To permit more flexible layout, and to solve some preprocessing problems (see §3.8.3), the Committee introduced string literal concatenation. Two string literals in a row are pasted together (with no null character in the middle) to make one combined string literal. This addition to the C language allows a programmer to extend a string literal beyond the end of a physical line without having to use the backslash-newline mechanism and thereby destroying the indentation scheme of the program. An explicit concatenation operator was not introduced because the concatenation is a lexical construct rather than a run-time operation.

without concatenation:

```
/* say the column is this wide */
        alpha = "abcdefghijklm\
nopqrstuvwxyz" ;
```

with concatenation:

```
/* say the column is this wide */
      alpha = "abcdefghijklm"
              "nopqrstuvwxyz";
```

QUIET CHANGE

A string of the form "\078" is valid, but now has different meaning. (See §3.1.3.)

QUIET CHANGE

A string of the form "\a" or "\x" now has different meaning. (See §3.1.3.)

QUIET CHANGE

It is neither required nor forbidden that identical string literals be represented by a single copy of the string in memory; a program depending upon either scheme may behave differently.

An **L** prefix distinguishes wide string literals. A prefix (as opposed to suffix) notation was adopted so that a translator can know at the start of the processing of a long string literal whether it is dealing with ordinary or wide characters. (See §2.2.1.2.)

3.1.5   Operators

Assignment operators of the form =+, described as *old fashioned* even in K&R, have been dropped.

The form += is now defined to be a single token, not two, so no white space is permitted within it; no compelling case could be made for permitting such white space.

QUIET CHANGE

Expressions of the form x=-3 change meaning with the loss of the old-style assignment operators.

The operator # has been added in preprocessing statements: within a #define it causes the macro argument following to be converted to a string literal.

The operator ## has also been added in preprocessing statements: within a #define it causes the tokens on either side to be *pasted* to make a single new token. See §3.8.3 for further discussion of these preprocessing operators.

3.1.6 Punctuators

The punctuator ... (ellipsis) has been added to denote a variable number of trailing arguments in a function prototype. (See §3.5.4.3.)

The constraint that certain punctuators must occur in pairs (and the similar constraint on certain operators in §3.1.5) only applies after preprocessing. Syntactic constraints are checked during syntactic analysis, and this follows preprocessing.

3.1.7 Header names

Header names in `#include` directives obey distinct tokenization rules; hence they are identified as distinct tokens. Attempting to treat quote-enclosed header names as string literals creates a contorted description of preprocessing, and the problems of treating angle-bracket-enclosed header names as a sequence of C tokens is even more severe.

3.1.8 Preprocessing numbers

The notion of preprocessing numbers has been introduced to simplify the description of preprocessing. It provides a means of talking about the tokenization of strings that look like numbers, or initial substrings of numbers, prior to their semantic interpretation. In the interests of keeping the description simple, occasional spurious forms are scanned as preprocessing numbers — `0x123E+1` is a single token under the rules. The Committee felt that it was better to tolerate such anomalies than burden the preprocessor with a more exact, and exacting, lexical specification. It felt that this anomaly was no worse than the principle under which the characters `a+++++b` are tokenized as `a ++ ++ + b` (an invalid expression), even though the tokenization `a ++ + ++ b` would yield a syntactically correct expression. In both cases, exercise of reasonable precaution in coding style avoids surprises.

3.1.9 Comments

The Committee considered proposals to allow comments to nest. The main argument for nesting comments is that it would allow programmers to "comment out" code. The Committee rejected this proposal on the grounds that comments should be used for adding documentation to a program, and that preferable mechanisms already exist for source code exclusion. For example,

```
#if 0
/* this code is bracketed out because ... */
code_to_be_excluded();
#endif
```

Preprocessing directives such as this prevent the enclosed code from being scanned by later translation phases. Bracketed material can include comments and other, nested, regions of bracketed code.

Another way of accomplishing these goals is with an `if` statement:

```
if (0) {
    /* this code is bracketed out because ... */
    code_to_be_excluded();
}
```

Many modern compilers will generate no code for this `if` statement.

3.2   Conversions

3.2.1   Arithmetic operands

3.2.1.1   Characters and integers

Since the publication of K&R, a serious divergence has occurred among implementations of C in the evolution of integral promotion rules. Implementations fall into two major camps, which may be characterized as *unsigned preserving* and *value preserving*. The difference between these approaches centers on the treatment of `unsigned char` and `unsigned short`, when widened by the *integral promotions*, but the decision has an impact on the typing of constants as well (see §3.1.3.2).

The *unsigned preserving* approach calls for promoting the two smaller unsigned types to `unsigned int`. This is a simple rule, and yields a type which is independent of execution environment.

The *value preserving* approach calls for promoting those types to `signed int`, if that type can properly represent all the values of the original type, and otherwise for promoting those types to `unsigned int`. Thus, if the execution environment represents `short` as something smaller than `int`, `unsigned short` becomes `int`; otherwise it becomes `unsigned int`.

Both schemes give the same answer in the vast majority of cases, and both give the same effective result in even more cases in implementations with two-complement arithmetic and quiet wraparound on signed overflow — that is, in most current implementations. In such implementations, differences between the two only appear when these two conditions are both true:

1. An expression involving an `unsigned char` or `unsigned short` produces an `int`-wide result in which the sign bit is set: i.e., either a unary operation on such a type, or a binary operation in which the other operand is an `int` or "narrower" type.

2. The result of the preceding expression is used in a context in which its signedness is significant:

   - `sizeof(int)` < `sizeof(long)` and it is in a context where it must be widened to a long type, or

- it is the left operand of the right-shift operator (in an implementation where this shift is defined as arithmetic), or
- it is either operand of /, %, <, <=, >, or >=.

In such circumstances a genuine ambiguity of interpretation arises. The result must be dubbed *questionably signed*, since a case can be made for either the signed or unsigned interpretation. Exactly the same ambiguity arises whenever an `unsigned int` confronts a `signed int` across an operator, and the `signed int` has a negative value. (Neither scheme does any better, or any worse, in resolving the ambiguity of this confrontation.) Suddenly, the negative signed `int` becomes a very large `unsigned int`, which may be surprising — or it may be exactly what is desired by a knowledgable programmer. Of course, *all of these ambiguities can be avoided by a judicious use of casts.*

One of the important outcomes of exploring this problem is the understanding that high-quality compilers might do well to look for such questionable code and offer (optional) diagnostics, and that conscientious instructors might do well to warn programmers of the problems of implicit type conversions.

The unsigned preserving rules greatly increase the number of situations where `unsigned int` confronts `signed int` to yield a questionably signed result, whereas the value preserving rules minimize such confrontations. Thus, the value preserving rules were considered to be safer for the novice, or unwary, programmer. After much discussion, the Committee decided in favor of value preserving rules, despite the fact that the UNIX C compilers had evolved in the direction of unsigned preserving.

QUIET CHANGE

A program that depends upon unsigned preserving arithmetic conversions will behave differently, probably without complaint. This is considered the most serious semantic change made by the Committee to a widespread current practice.

The Standard clarifies that the integral promotion rules also apply to bit-fields.

3.2.1.2 Signed and unsigned integers

Precise rules are now provided for converting to and from unsigned integers. On a twos-complement machine, the operation is still virtual (no change of representation is required), but the rules are now stated independent of representation.

3.2.1.3 Floating and integral

There was strong agreement that floating values should truncate toward zero when converted to an integral type, the specification adopted in the Standard. Although the Base Document permitted negative floating values to truncate away from zero, no Committee member knew of current hardware that functions in such a manner.[2]

---

[2] We have since been informed of one such implementation.

3.2.1.4   Floating types

The Standard, unlike the Base Document, does not require rounding in the `double` to `float` conversion. Some widely used IEEE floating point processor chips control floating to integral conversion with the same mode bits as for double-precision to single-precision conversion; since truncation-toward-zero is the appropriate setting for C in the former case, it would be expensive to require such implementations to round to `float`.

3.2.1.5   Usual arithmetic conversions

The rules in the Standard for these conversions are slight modifications of those in the Base Document: the modifications accommodate the added types and the value preserving rules (see §3.2.1.1). Explicit license has been added to perform calculations in a "wider" type than absolutely necessary, since this can sometimes produce smaller and faster code (not to mention the correct answer more often). Calculations can also be performed in a "narrower" type, by the *as if* rule, so long as the same end result is obtained.   *Explicit casting can always be used to obtain exactly the intermediate types required.*

The Committee relaxed the requirement that `float` operands be converted to `double`. An implementation may still choose to convert.

<div align="center">QUIET CHANGE</div>

Expressions with `float` operands may now be computed at lower precision. The Base Document specified that all floating point operations be done in `double`.

3.2.2   Other operands

3.2.2.1   Lvalues and function designators

A difference of opinion within the C community has centered around the meaning of *lvalue*, one group considering an lvalue to be any kind of object locator, another group holding that an lvalue is meaningful on the left side of an assigning operator. The Committee has adopted the definition of lvalue as an object locator. The term *modifiable lvalue* is used for the second of the above concepts.

The role of array objects has been a classic source of confusion in C, in large part because of the numerous contexts in which an array reference is converted to a pointer to its first element. While this conversion neatly handles the semantics of subscripting, the fact that `a[i]` is itself a modifiable lvalue while `a` is not has puzzled many students of the language. A more precise description has therefore been incorporated in the Standard, in the hopes of combatting this confusion.

3.2.2.2 `void`

The description of operators and expressions is simplified by saying that `void` yields a value, with the understanding that the value has no representation, hence requires no storage.

3.2.2.3 Pointers

C has now been implemented on a wide range of architectures. While some of these architectures feature uniform pointers which are the size of some integer type, maximally portable code may not assume any necessary correspondence between different pointer types and the integral types.

The use of `void *` ("pointer to void") as a generic object pointer type is an invention of the Committee. Adoption of this type was stimulated by the desire to specify function prototype arguments that either quietly convert arbitrary pointers (as in `fread`) or complain if the argument type does not exactly match (as in `strcmp`). Nothing is said about pointers to functions, which may be incommensurate with object pointers and/or integers.

Since pointers and integers are now considered incommensurate, the only integer that can be safely converted to a pointer is the constant 0. The result of converting any other integer to a pointer is machine dependent.

Consequences of the treatment of pointer types in the Standard include:

- A pointer to void may be converted to a pointer to an object of any type.

- A pointer to any object of any type may be converted to a pointer to void.

- If a pointer to an object is converted to a pointer to void and back again to the original pointer type, the result compares equal to original pointer.

- It is invalid to convert a pointer to an object of any type to a pointer to an object of a different type without an explicit cast.

- Even with an explicit cast, it is invalid to convert a function pointer to an object pointer or a pointer to void, or vice-versa.

- It is invalid to convert a pointer to a function of one type to a pointer to a function of a different type without a cast.

- Pointers to functions that have different parameter-type information (including the "old-style" absence of parameter-type information) are different types.

Implicit in the Standard is the notion of *invalid pointers*. In discussing pointers, the Standard typically refers to "a pointer to an object" or "a pointer to a function" or "a null pointer." A special case in address arithmetic allows for a pointer to just past the end of an array. Any other pointer is invalid.

RATIONALE

An invalid pointer might be created in several ways. An arbitrary value can be assigned (via a cast) to a pointer variable. (This could even create a valid pointer, depending on the value.) A pointer to an object becomes invalid if the memory containing the object is deallocated. Pointer arithmetic can produce pointers outside the range of an array.

Regardless how an invalid pointer is created, any use of it yields undefined behavior. Even assignment, comparison with a null pointer constant, or comparison with itself, might on some systems result in an exception.

Consider a hypothetical segmented architecture, on which pointers comprise a segment descriptor and an offset. Suppose that segments are relatively small, so that large arrays are allocated in multiple segments. While the segments are valid (allocated, mapped to real memory), the hardware, operating system, or C implementation can make these multiple segments behave like a single object: pointer arithmetic and relational operators use the defined mapping to impose the proper order on the elements of the array. Once the memory is deallocated, the mapping is no longer guaranteed to exist; use of the segment descriptor might now cause an exception, or the hardware addressing logic might return meaningless data.

3.3   Expressions

Several closely-related topics are involved in the precise specification of expression evaluation: *precedence*, *associativity*, *grouping*, *sequence points*, *agreement points*, *order of evaluation*, and *interleaving*. The latter three terms are discussed in §2.1.2.3.

The rules of *precedence* are encoded into the syntactic rules for each operator. For example, the syntax for *additive-expression* includes the rule

> *additive-expression* **+** *multiplicative-expression*

which implies that a+b*c parses as a+(b*c). The rules of *associativity* are similarly encoded into the syntactic rules. For example, the syntax for *assignment-expression* includes the rule

> *unary-expression assignment-operator assignment-expression*

which implies that a=b=c parses as a=(b=c).

With rules of precedence and associativity thus embodied in the syntax rules, the Standard specifies, in general, the *grouping* (association of operands with operators) in an expression.

The Base Document describes C as a language in which the operands of successive identical commutative associative operators can be regrouped. The Committee has decided to remove this license from the Standard, thus bringing C into accord with most other major high-level languages.

This change was motivated primarily by the desire to make C more suitable for floating point programming. Floating point arithmetic does not obey many of the mathematical rules that real arithmetic does. For instance, the two expressions

(a+b)+c and a+(b+c) may well yield different results: suppose that b is greater than 0, a equals -b, and c is positive but substantially smaller than b. (That is, suppose c/b is less than DBL_EPSILON.) Then (a+b)+c is 0+c, or c, while a+(b+c) equals a+b, or 0. That is to say, floating point addition (and multiplication) is not associative.

The Base Document's rule imposes a high cost on translation of numerical code to C. Much numerical code is written in FORTRAN, which does provide a no-regrouping guarantee; indeed, this is the normal semantic interpretation in most high-level languages other than C. The Base Document's advice, "rewrite using explicit temporaries," is burdensome to those with tens or hundreds of thousands of lines of code to convert, a conversion which in most other respects could be done automatically.

Elimination of the regrouping rule does not in fact prohibit much regrouping of integer expressions. The bitwise logical operators can be arbitrarily regrouped, since any regrouping gives the same result *as if* the expression had not been re-grouped. This is also true of integer addition and multiplication in implementations with twos-complement arithmetic and silent wraparound on overflow. Indeed, in any implementation, regroupings which do not introduce overflows behave *as if* no regrouping had occurred. (Results may also differ in such an implementation if the expression as written results in overflows: in such a case the behavior is undefined, so any regrouping couldn't be any worse.)

The types of lvalues that may be used to access an object have been restricted so that an optimizer is not required to make worst-case aliasing assumptions.

In practice, aliasing arises with the use of pointers. A contrived example to illustrate the issues is

```
int a;

void f(int * b)
{
    a = 1;
    *b = 2;
    g(a);
}
```

It is tempting to generate the call to g as if the source expression were g(1), but b might point to a, so this optimization is not safe. On the other hand, consider

```
int a;
void f( double * b )
{
    a = 1;
    *b = 2.0;
    g(a);
}
```

Again the optimization is incorrect only if **b** points to **a**. However, this would only have come about if the address of **a** were somewhere cast to (**double***). The Committee has decided that such dubious possibilities need not be allowed for.

In principle, then, aliasing only need be allowed for when the lvalues all have the same type. In practice, the Committee has recognized certain prevalent exceptions:

- The lvalue types may differ in signedness. In the common range, a signed integral type and its unsigned variant have the same representation; it was felt that an appreciable body of existing code is not "strictly typed" in this area.

- Character pointer types are often used in the bytewise manipulation of objects; a byte stored through such a character pointer may well end up in an object of any type.

- A qualified version of the object's type, though formally a different type, provides the same interpretation of the value of the object.

Structure and union types also have problematic aliasing properties:

```
struct fi{ float f; int i;};

void f( struct fi * fip, int * ip )
{
    static struct fi a = {2.0, 1};
    *ip = 2;
    *fip = a;
    g(*ip);

    *fip = a;
    *ip = 2;
    g(fip->i);
}
```

It is not safe to optimize the first call to **g** as **g(2)**, or the second as **g(1)**, since the call to **f** could quite legitimately have been

```
struct fi  x;
f( &x, &x.i );
```

These observations explain the other exception to the same-type principle.

3.3.1   Primary expressions

A primary expression may be **void** (parenthesized call to a function returning **void**), a function designator (identifier or parenthesized function designator), an lvalue (identifier or parenthesized lvalue), or simply a value expression. Constraints ensure

that a `void` primary expression is no part of a further expression, except that a void expression may be cast to void, may be the second or third operand of a conditional operator, or may be an operand of a comma operator.

3.3.2 Postfix operators

3.3.2.1 Array subscripting

The Committee found no reason to disallow the symmetry that permits `a[i]` to be written as `i[a]`.

The syntax and semantics of multidimensional arrays follow logically from the definition of arrays and the subscripting operation. The material in the Standard on multidimensional arrays introduces no new language features, but clarifies the C treatment of this important abstract data type.

3.3.2.2 Function calls

Pointers to functions may be used either as `(*pf)()` or as `pf()`. The latter construct, not sanctioned in the Base Document, appears in some present versions of C, is unambiguous, invalidates no old code, and can be an important shorthand. The shorthand is useful for packages that present only one external name, which designates a structure full of pointers to objects and functions: member functions can be called as `graphics.open(file)` instead of `(*graphics.open)(file)`.

The treatment of function designators can lead to some curious, but valid, syntactic forms. Given the declarations:

```
int f(), (*pf)();
```

then all of the following expressions are valid function calls:

```
(&f)(); f(); (*f)(); (**f)(); (***f)();
pf();   (*pf)(); (**pf)(); (***pf)();
```

The first expression on each line was discussed in the previous paragraph. The second is conventional usage. All subsequent expressions take advantage of the implicit conversion of a function designator to a pointer value, in nearly all expression contexts. The Committee saw no real harm in allowing these forms; outlawing forms like `(*f)()`, while still permitting `*a` (for `int a[]`), simply seemed more trouble than it was worth.

The rule for implicit declaration of functions has been retained, but various past ambiguities have been resolved by describing this usage in terms of a corresponding explicit declaration.

For compatibility with past practice, all argument promotions occur as described in the Base Document in the absence of a prototype declaration, including the (not always desirable) promotion of `float` to `double`. A prototype gives the implementor explicit license to pass a `float` as a `float` rather than a `double`, or a `char` as a

`char` rather than an `int`, or an argument in a special register, etc. If the definition of a function in the presence of a prototype would cause the function to expect other than the default promotion types, then clearly the calls to this function must be made in the presence of a compatible prototype.

To clarify this and other relationships between function calls and function definitions, the Standard describes an equivalence between a function call or definition which does occur in the presence of a prototype and one that does not.

Thus a prototyped function with no "narrow" types and no variable argument list must be callable in the absence of a prototype, since the types actually passed in a call are equivalent to the explicit function definition prototype. This constraint is necessary to retain compatibility with past usage of library functions. (See §4.1.3.)

This provision constrains the latitude of an implementor because the parameter passing conventions of prototype and non-prototype function calls must be the same for functions accepting a fixed number of arguments. Implementations in environments where efficient function calling mechanisms are available must, in effect, use the efficient calling sequence either in all "fixed argument list" calls or in none. Since efficient calling sequences often do not allow for variable argument functions, the fixed part of a variable argument list may be passed in a completely different fashion than in a fixed argument list with the same number and type of arguments.

The existing practice of omitting trailing parameters in a call if it is known that the parameters will not be used has consistently been discouraged. Since omission of such parameters creates an inequivalence between the call and the declaration, the behavior in such cases is undefined, and a maximally portable program will avoid this usage. Hence an implementation is free to implement a function calling mechanism for fixed argument lists which would (perhaps fatally) fail if the wrong number or type of arguments were to be provided.

Strictly speaking then, calls to `printf` are obliged to be in the scope of a prototype (as by `#include <stdio.h>`), but implementations are not obliged to fail on such a lapse. (The behavior is *undefined*).

3.3.2.3   Structure and union members

Since the language now permits structure parameters, structure assignment and functions returning structures, the concept of a *structure expression* is now part of the C language. A structure value can be produced by an assignment, by a function call, by a comma operator expression or by a conditional operator expression:

```
s1 = (s2 = s3)
sf(x)
(x, s1)
x ? s1 : s2
```

In these cases, the result is *not* an lvalue; hence it cannot be assigned to nor can its address be taken.

Similarly, `x.y` is an lvalue only if `x` is an lvalue. Thus none of the following valid expressions are lvalues:

```
sf(3).a
(s1=s2).a
((i==6)?s1:s2).a
(x,s1).a
```

Even when `x.y` is an lvalue, it may not be modifiable:

```
const struct S s1;
s1.a = 3;              /* invalid */
```

The Standard requires that an implementation diagnose a *constraint error* in the case that the member of a structure or union designated by the identifier following a member selection operator (`.` or `->`) does not appear in the type of the structure or union designated by the first operand. The Base Document is unclear on this point.

3.3.2.4   Postfix increment and decrement operators

The Committee has not endorsed the practice in some implementations of considering post-increment and post-decrement operator expressions to be lvalues.

3.3.3   Unary operators

3.3.3.1   Prefix increment and decrement operators

See §3.3.2.4.

3.3.3.2   Address and indirection operators

Some implementations have not allowed the `&` operator to be applied to an array or a function. (The construct was permitted in early versions of C, then later made optional.) The Committee has endorsed the construct since it is unambiguous, and since data abstraction is enhanced by allowing the important `&` operator to apply uniformly to any addressable entity.

3.3.3.3   Unary arithmetic operators

Unary plus was adopted by the Committee from several implementations, for symmetry with unary minus.

   The bitwise complement operator ˜, and the other bitwise operators, have now been defined arithmetically for unsigned operands. Such operations are well-defined because of the restriction of integral representations to "binary numeration systems." (See §3.1.2.5.)

3.3.3.4   The `sizeof` operator

It is fundamental to the correct usage of functions such as `malloc` and `fread` that `sizeof (char)` be exactly one. In practice, this means that a *byte* in C terms is the smallest unit of storage, even if this unit is 36 bits wide; and all objects are comprised of an integral number of these smallest units. (See §1.6.)

The Standard, like the Base Document, defines the result of the `sizeof` operator to be a constant of an unsigned integral type. Common implementations, and common usage, have often presumed that the resulting type is `int`. Old code that depends on this behavior has never been portable to implementations that define the result to be a type other than `int`. The Committee did not feel it was proper to change the language to protect incorrect code.

The type of `sizeof`, whatever it is, is published (in the library header `<stddef.h>`) as `size_t`, since it is useful for the programmer to be able to refer to this type. This requirement implicitly restricts `size_t` to be a synonym for an existing unsigned integer type, thus quashing any notion that the largest declarable object might be too big to span even with an `unsigned long`. This also restricts the maximum number of elements that may be declared in an array, since for any array `a` of `N` elements,

        N == sizeof(a)/sizeof(a[0])

Thus `size_t` is also a convenient type for array sizes, and is so used in several library functions. (See §4.9.8.1, §4.9.8.2, §4.10.3.1, etc.)

The Standard specifies that the argument to `sizeof` can be any value except a bit field, a void expression, or a function designator. This generality allows for interesting environmental enquiries; given the declarations

        int   *p, *q;

these expressions determine the size of the type used for ...

        sizeof(F(x))   /* ... F's return value */
        sizeof(p-q)   /* ... pointer difference */

(The last type is of course available as `ptrdiff_t` in `<stddef.h>`.)

3.3.4   Cast operators

A `(void)` cast is explicitly permitted, more for documentation than for utility.

Nothing portable can be said about casting integers to pointers, or vice versa, since the two are now incommensurate.

The definition of these conversions adopted in the Standard resembles that in the Base Document, but with several significant differences. The Base Document required that a pointer successfully converted to an integer must be guaranteed to

be convertible back to the same pointer. This integer-to-pointer conversion is now specified as *implementation-defined.* While a high-quality implementation would preserve the same address value whenever possible, it was considered impractical to require that the identical representation be preserved. The Committee noted that, on some current machine implementations, identical representations are required for efficient code generation for pointer comparisons and arithmetic operations.

The conversion of the integer constant 0 to a pointer is defined similarly to the Base Document. The resulting pointer must not address any object, must appear to be equal to an integer value of 0, and may be assigned to or compared for equality with any other pointer. This definition does not necessarily imply a representation by a bit pattern of all zeros: an implementation could, for instance, use some address which causes a hardware trap when dereferenced.

The type `char` must have the least strict alignment of any type, so `char *` has often been used as a portable type for representing arbitrary object pointers. This usage creates an unfortunate confusion between the ideas of *arbitrary pointer* and *character or string pointer.* The new type `void *`, which has the same representation as `char *`, is therefore preferable for arbitrary pointers.

It is possible to cast a pointer of some qualified type (§3.5.3) to an unqualified version of that type. Since the qualifier defines some special access or aliasing property, however, any dereference of the cast pointer results in *undefined behavior.*

The Standard (§3.2.1.4) requires that a cast of one floating point type to another (e.g., `double` to `float`) results in an actual conversion.

3.3.5 Multiplicative operators

There was considerable sentiment for giving more portable semantics to division (and hence remainder) by specifying some way of giving less machine dependent results for negative operands. Few Committee members wanted to require this by default, lest existing fast code be gravely slowed. One suggestion was to make `signed int` a type distinct from plain `int`, and require better-defined semantics for `signed int` division and remainder. This suggestion was opposed on the grounds that effectively adding several types would have consequences out of proportion to the benefit to be obtained; the Committee twice rejected this approach. Instead the Committee has adopted new library functions `div` and `ldiv` which produce integral quotient and remainder with well-defined sign semantics. (See §4.10.6.2, §4.10.6.3.)

The Committee rejected extending the `%` operator to work on floating types; such usage would duplicate the facility provided by `fmod`. (See §4.5.6.5.)

3.3.6 Additive operators

As with the `sizeof` operator, implementations have taken different approaches in defining a type for the difference between two pointers (see §3.3.3.4). It is important

that this type be signed, in order to obtain proper algebraic ordering when dealing with pointers within the same array. However, the magnitude of a pointer difference can be as large as the size of the largest object that can be declared. (And since that is an unsigned type, the difference between two pointers may cause an overflow.)

The type of *pointer minus pointer* is defined to be `int` in K&R. The Standard defines the result of this operation to be a signed integer, the size of which is implementation-defined. The type is published as `ptrdiff_t`, in the standard header `<stddef.h>`. Old code recompiled by a conforming compiler may no longer work if the implementation defines the result of such an operation to be a type other than `int` and if the program depended on the result to be of type `int`. This behavior was considered by the Committee to be correctable. Overflow was considered not to break old code since it was undefined by K&R. Mismatch of types between actual and formal argument declarations is correctable by including a properly defined function prototype in the scope of the function invocation.

An important endorsement of widespread practice is the requirement that a pointer can always be incremented to *just past* the end of an array, with no fear of overflow or wraparound:

```
SOMETYPE array[SPAN];
/* ... */
for (p = &array[0]; p < &array[SPAN]; p++)
```

This stipulation merely requires that every object be followed by one byte whose address is representable. That byte can be the first byte of the next object declared for all but the last object located in a contiguous segment of memory. (In the example, the address `&array[SPAN]` must address a byte following the highest element of `array`.) Since the pointer expression `p+1` need not (and should not) be dereferenced, it is unnecessary to leave room for a complete object of size `sizeof(*p)`.

In the case of `p-1`, on the other hand, an entire object *would* have to be allocated prior to the array of objects that `p` traverses, so decrement loops that run off the bottom of an array may fail. This restriction allows segmented architectures, for instance, to place objects at the start of a range of addressable memory.

3.3.7   Bitwise shift operators

See §3.3.3.3 for a discussion of the arithmetic definition of these operators.

The description of shift operators in K&R suggests that shifting by a `long` count should force the left operand to be widened to `long` before being shifted. A more intuitive practice, endorsed by the Committee, is that the type of the shift count has no bearing on the type of the result.

QUIET CHANGE

Shifting by a `long` count no longer coerces the shifted operand to `long`.

The Committee has affirmed the freedom in implementation granted by the Base Document in not requiring the signed right shift operation to sign extend, since such a requirement might slow down fast code and since the usefulness of sign extended shifts is marginal. (Shifting a negative twos-complement integer arithmetically right one place is *not* the same as dividing by two!)

3.3.8 Relational operators

For an explanation of why the pointer comparison of the object pointer P with the pointer expression P+1 is always safe, see Rationale §3.3.6.

3.3.9 Equality operators

The Committee considered, on more than one occasion, permitting comparison of structures for equality. Such proposals foundered on the problem of holes in structures. A byte-wise comparison of two structures would require that the holes assuredly be set to zero so that all holes would compare equal, a difficult task for automatic or dynamically allocated variables. (The possibility of union-type elements in a structure raises insuperable problems with this approach.) Otherwise the implementation would have to be prepared to break a structure comparison into an arbitrary number of member comparisons; a seemingly simple expression could thus expand into a substantial stretch of code, which is contrary to the *spirit of C.*

In pointer comparisons, one of the operands may be of type void *. In particular, this allows NULL, which can be defined as (void *)0, to be compared to any object pointer.

3.3.10 Bitwise AND operator

See §3.3.3.3 for a discussion of the arithmetic definition of the bitwise operators.

3.3.11 Bitwise exclusive OR operator

See §3.3.3.3.

3.3.12 Bitwise inclusive OR operator

See §3.3.3.3.

3.3.13 Logical AND operator

3.3.14 Logical OR operator

3.3.15 Conditional operator

The syntactic restrictions on the middle operand of the conditional operator have been relaxed to include more than just *logical-OR-expression*: several extant implementations have adopted this practice.

The type of a conditional operator expression can be `void`, a structure, or a union; most other operators do not deal with such types. The rules for balancing type between pointer and integer have, however, been tightened, since now only the constant 0 can portably be coerced to pointer.

The Standard allows one of the second or third operands to be of type `void *`, if the other is a pointer type. Since the result of such a conditional expression is `void *`, an appropriate cast must be used.

3.3.16   Assignment operators

Certain syntactic forms of assignment operators have been discontinued, and others tightened up (see §3.1.5).

The storage assignment need not take place until the next sequence point. (A restriction in earlier drafts that the storage take place before the value of the expression is used has been removed.) As a consequence, a straightforward syntactic test for ambiguous expressions can be stated. Some definitions: A *side effect* is a storage to any data object, or a read of a volatile object. An *ambiguous expression* is one whose value depends upon the order in which side effects are evaluated. A *pure function* is one with no side effects; an impure function is any other. A *sequenced expression* is one whose major operator defines a sequence point: comma, `&&`, `||`, or conditional operator; an *unsequenced expression* is any other. We can then say that an unsequenced expression is ambiguous if more than one operand invokes any impure function, or if more than one operand contains an lvalue referencing the same object and one or more operands specify a side-effect to that object. Further, any expression containing an ambiguous expression is ambiguous.

The optimization rules for factoring out assignments can also be stated. Let `X(i,S)` be an expression which contains no impure functions or sequenced operators, and suppose that `X` contains a storage `S(i)` to `i`. The storage expressions, and related expressions, are

```
S(i):       Sval(i):       Snew(i):
++i         i+1            i+1
i++         i              i+1
--i         i-1            i-1
i--         i              i-1
i = y       y              y
i op= y     i op y         i op y
```

Then `X(i,S)` can be replaced by either

```
(T = i, i = Snew(i), X(T,Sval))
```

or

```
(T = X(i,Sval), i = Snew(i), T)
```

provided that neither `i` nor `y` have side effects themselves.

3.3.16.1   Simple assignment

Structure assignment has been added: its use was foreshadowed even in K&R, and many existing implementations already support it.

The rules for type compatibility in assignment also apply to argument compatibility between actual argument expressions and their corresponding argument types in a function prototype.

An implementation need not correctly perform an assignment between overlapping operands. Overlapping operands occur most naturally in a union, where assigning one field to another is often desirable to effect a type conversion in place; the assignment may well work properly in all simple cases, but it is not maximally portable. Maximally portable code should use a temporary variable as an intermediate in such an assignment.

3.3.16.2   Compound assignment

The importance of requiring that the left operand lvalue be evaluated only once is not a question of efficiency, although that is one compelling reason for using the compound assignment operators. Rather, it is to assure that any side effects of evaluating the left operand are predictable.

3.3.17   Comma operator

The left operand of a comma operator may be void, since only the right-hand operator is relevant to the type of the expression.

The example in the Standard clarifies that commas separating arguments "bind" tighter than the comma operator in expressions.

3.4   Constant Expressions

To clarify existing practice, several varieties of constant expression have been identified:

The expression following #if (§3.8.1) must expand to integer constants, character constants, the special operator defined, and operators with no side effects. No environmental inquiries can be made, since all arithmetic is done as translate-time (signed or unsigned) long integers, and casts are disallowed. The restriction to translate-time arithmetic frees an implementation from having to perform execution-environment arithmetic in the host environment. It does not preclude an implementation from doing so — the implementation may simply define "translate-time arithmetic" to be that of the target.

Unsigned arithmetic is performed in these expressions (according to the default widening rules) when unsigned operands are involved; this rule allows for unsurprising arithmetic involving very large constants (i.e, those whose type is unsigned

`long`) since they cannot be represented as `long` or constants explicitly marked as unsigned.

Character constants, when evaluated in `#if` expressions, may be interpreted in the source character set, the execution character set, or some other implementation-defined character set. This latitude reflects the diversity of existing practice, especially in cross-compilers.

An *integral constant expression* must involve only numbers knowable at translate time, and operators with no side effects. Casts and the `sizeof` operator may be used to interrogate the execution environment.

Static initializers include integral constant expressions, along with floating constants and simple addressing expressions. An implementation must accept arbitrary expressions involving floating and integral numbers and side-effect-free operators in arithmetic initializers, but it is at liberty to turn such initializers into executable code which is invoked prior to program startup (see §2.1.2.2); this scheme might impose some requirements on linkers or runtime library code in some implementations.

The translation environment must not produce a less accurate value for a floating-point initializer than the execution environment, but it is at liberty to do better. Thus a static initializer may well be slightly different than the same expression computed at execution time. However, while implementations are certainly *permitted* to produce exactly the same result in translation and execution environments, *requiring* this was deemed to be an intolerable burden on many cross-compilers.

QUIET CHANGE

A program that uses `#if` expressions to determine properties of the execution environment may now get different answers.

3.5   Declarations

The Committee decided that empty declarations are invalid (except for a special case with tags, see §3.5.2.3, and the case of enumerations such as `enum {zero,one};`, see §3.5.2.2). While many seemingly silly constructs are tolerated in other parts of the language in the interest of facilitating the machine generation of C, empty declarations were considered sufficiently easy to avoid.

The practice of placing the storage class specifier other than first in a declaration has been branded as *obsolescent* (See §3.9.3.) The Committee feels it desirable to rule out such constructs as

```
enum { aaa, aab,
    /* etc */
zzy, zzz } typedef a2z;
```

in some future standard.

3.5.1   Storage-class specifiers

Because the address of a `register` variable cannot be taken, objects of storage class `register` effectively exist in a space distinct from other objects. (Functions occupy yet a third address space). This makes them candidates for optimal placement, the usual reason for declaring registers, but it also makes them candidates for more aggressive optimization.

The practice of representing register variables as wider types (as when `register char` is quietly changed to `register int`) is no longer acceptable.

3.5.2   Type specifiers

Several new type specifiers have been added: `signed`, `enum`, and `void`. `long float` has been retired and `long double` has been added, along with a plethora of integer types. The Committee's reasons for each of these additions, and the one deletion, are given in section §3.1.2.5 of this document.

3.5.2.1   Structure and union specifiers

Three types of bit fields are now defined: "plain" `int` calls for *implementation-defined* signedness (as in the Base Document), `signed int` calls for assuredly signed fields, and `unsigned int` calls for unsigned fields. The old constraints on bit fields crossing *word* boundaries have been relaxed, since so many properties of bit fields are implementation dependent anyway.

The layout of structures is determined only to a limited extent:

- no hole may occur at the beginning;

- members occupy increasing storage addresses; and

- if necessary, a hole is placed on the end to make the structure big enough to pack tightly into arrays and maintain proper alignment.

Since some existing implementations, in the interest of enhanced access time, leave internal holes larger than absolutely necessary, it is not clear that a portable deterministic method can be given for traversing a structure field by field.

To clarify what is meant by the notion that "all the fields of a union occupy the same storage," the Standard specifies that a pointer to a union, when suitably cast, points to each member (or, in the case of a bit-field member, to the storage unit containing the bit field).

3.5.2.2   Enumeration specifiers

3.5.2.3   Tags

As with all block structured languages that also permit forward references, C has a problem with structure and union tags. If one wants to declare, within a block, two mutually referencing structures, one must write something like:

```
struct x { struct y *p; /*...*/ };
struct y { struct x *q; /*...*/ };
```

But if `struct y` is already defined in a containing block, the first field of `struct x` will refer to the older declaration.

Thus special semantics has been given to the form:

```
struct y;
```

It now hides the outer declaration of `y`, and "opens" a new instance in the current block.

QUIET CHANGE

The empty declaration `struct x;` is no longer innocuous.

3.5.3   Type qualifiers

The Committee has added to C two *type qualifiers*: `const` and `volatile`. Individually and in combination they specify the assumptions a compiler can and must make when accessing an object through an lvalue.

The syntax and semantics of `const` were adapted from C++; the concept itself has appeared in other languages. `volatile` is an invention of the Committee; it follows the syntactic model of `const`.

Type qualifiers were introduced in part to provide greater control over optimization. Several important optimization techniques are based on the principle of "cacheing": under certain circumstances the compiler can remember the last value accessed (read or written) from a location, and use this retained value the next time that location is read. (The memory, or "cache", is typically a hardware register.) If this memory is a machine register, for instance, the code can be smaller and faster using the register rather than accessing external memory.

The basic qualifiers can be characterized by the restrictions they impose on access and cacheing:

`const` No writes through this lvalue. In the absence of this qualifier, writes may occur through this lvalue.

`volatile` No cacheing through this lvalue: each operation in the abstract semantics must be performed. (That is, no cacheing assumptions may be made, since the location is not guaranteed to contain any previous value.) In the absence of this qualifier, the contents of the designated location may be assumed to be unchanged (except for possible aliasing.)

A translator design with no cacheing optimizations can effectively ignore the type qualifiers, except insofar as they affect assignment compatibility.

It would have been possible, of course, to specify a `nonconst` keyword instead of `const`, or `nonvolatile` instead of `volatile`. The senses of these concepts in

the Standard were chosen to assure that the default, unqualified, case was the most common, and that it corresponded most clearly to traditional practice in the use of lvalue expressions.

Four combinations of the two qualifiers is possible; each defines a useful set of lvalue properties. The next several paragraphs describe typical uses of these qualifiers.

The translator may assume, for an unqualified lvalue, that it may read or write the referenced object, that the value of this object cannot be changed except by explicitly programmed actions in the current thread of control, but that other lvalue expressions could reference the same object.

const is specified in such a way that an implementation is at liberty to put const objects in read-only storage, and is encouraged to diagnose obvious attempts to modify them, but is not required to track down all the subtle ways that such checking can be subverted. If a function parameter is declared const, then the referenced object is not changed (through that lvalue) in the body of the function — the parameter is read-only.

A static volatile object is an appropriate model for a memory-mapped I/O register. Implementors of C translators should take into account relevant hardware details on the target systems when implementing accesses to volatile objects. For instance, the hardware logic of a system may require that a two-byte memory-mapped register not be accessed with byte operations; a compiler for such a system would have to assure that no such instructions were generated, even if the source code only accesses one byte of the register. Whether read-modify-write instructions can be used on such device registers must also be considered. Whatever decisions are adopted on such issues must be documented, as volatile access is implementation-defined. A volatile object is an appropriate model for a variable shared among multiple processes.

A static const volatile object appropriately models a memory-mapped input port, such as a real-time clock. Similarly, a const volatile object models a variable which can be altered by another process but not by this one.

Although the type qualifiers are formally treated as defining new types they actually serve as modifiers of declarators. Thus the declarations

```
const struct s {int a,b;} x;
struct s   y;
```

declare x as a const object, but not y. The const property can be associated with the aggregate type by means of a type definition:

```
typedef const struct s {int a,b;} stype;
stype x;
stype y;
```

In these declarations the const property is associated with the declarator stype, so x and y are both const objects.

The Committee considered making `const` and `volatile` storage classes, but this would have ruled out any number of desirable constructs, such as `const` members of structures and variable pointers to `const` types.

A cast of a value to a qualified type has no effect; the qualification (`volatile`, say) can have no effect on the access since it has occurred prior to the cast. If it is necessary to access a non-volatile object using volatile semantics, the technique is to cast the address of the object to the appropriate pointer-to-qualified type, then dereference that pointer.

3.5.4   Declarators

The function prototype syntax was adapted from C++. (See §3.3.2.2 and §3.5.4.3)

Some current implementations have a limit of six type modifiers (*function returning, array of, pointer to*), the limit used in Ritchie's original compiler. This limit has been raised to twelve since the original limit has proven insufficient in some cases; in particular, it did not allow for FORTRAN-to-C translation, since FORTRAN allows for seven subscripts. (Some users have reported using nine or ten levels, particularly in machine-generated C code.)

3.5.4.1   Pointer declarators

A pointer declarator may have its own type qualifiers, to specify the attributes of the pointer itself, as opposed to those of the reference type. The construct is adapted from C++.

`const int *` means *(variable) pointer to constant* `int`, and `int * const` means *constant pointer to (variable)* `int`, just as in C++, from which these constructs were adopted. (And *mutatis mutandis* for the other type qualifiers.) As with other aspects of C type declarators, judicious use of `typedef` statements can clarify the code.

3.5.4.2   Array declarators

The concept of *composite types* (§3.1.2.6) was introduced to provide for the accretion of information from incomplete declarations, such as array declarations with missing size, and function declarations with missing prototype (argument declarations). Type declarators are therefore said to specify *compatible types* if they agree except for the fact that one provides less information of this sort than the other.

The declaration of 0-length arrays is invalid, under the general principle of not providing for 0-length objects. The only common use of this construct has been in the declaration of dynamically allocated variable-size arrays, such as

```
struct segment {
    short int count;
    char c[N];
};
```

```
struct segment * new_segment( const int length )
{
    struct segment * result;
    result = malloc( sizeof segment + (length-N) );
    result->count = length;
    return result;
}
```

In such usage, N would be 0 and (length-N) would be written as length. But this paradigm works just as well, as written, if N is 1. (Note, by the by, an alternate way of specifying the size of result:

```
result = malloc( offsetof(struct segment,c) + length );
```

This illustrates one of the uses of the offsetof macro.)

3.5.4.3 Function declarators (including prototypes)

The function prototype mechanism is one of the most useful additions to the C language. The feature, of course, has precedent in many of the Algol-derived languages of the past 25 years. The particular form adopted in the Standard is based in large part upon C++.

Function prototypes provide a powerful translation-time error detection capability. In traditional C practice without prototypes, it is extremely difficult for the translator to detect errors (wrong number or type of arguments) in calls to functions declared in another source file. Detection of such errors has either occurred at runtime, or through the use of auxiliary software tools.

In function calls not in the scope of a function prototype, integral arguments have the *integral widening conversions* applied and float arguments are widened to double. It is thus impossible in such a call to pass an unconverted char or float argument. Function prototypes give the programmer explicit control over the function argument type conversions, so that the often inappropriate and sometimes inefficient default widening rules for arguments can be suppressed by the implementation. Modifications of function interfaces are easier in cases where the actual arguments are still assignment compatible with the new formal parameter type — only the function definition and its prototype need to be rewritten in this case; no function calls need be rewritten.

Allowing an optional identifier to appear in a function prototype serves two purposes:

- the programmer can associate a meaningful name with each argument position for documentation purposes, and

- a function declarator and a function prototype can use the same syntax. The consistent syntax makes it easier for new users of C to learn the language. Automatic generation of function prototype declarators from function definitions is also facilitated.

Optimizers can also take advantage of function prototype information. Consider this example:

```
extern int compare(const char * string1,
                   const char * string2) ;

void func2(int x)
{
        char * str1, * str2 ;
            /* ... */
        x = compare(str1, str2) ;
            /* ... */
}
```

The optimizer knows that the pointers passed to **compare** are not used to assign new values to any objects that the pointers reference. Hence the optimizer can make less conservative assumptions about the side effects of **compare** than would otherwise be necessary.

The Standard requires that calls to functions taking a variable number of arguments must occur in the presence of a prototype (using the trailing ellipsis notation ,...). An implementation may thus assume that all other functions are called with a fixed argument list, and may therefore use possibly more efficient calling sequences. Programs using old-style headers in which the number of arguments in the calls and the definition differ may not work in implementations which take advantage of such optimizations. This is not a Quiet Change, strictly speaking, since the program does not conform to the Standard. A word of warning is in order, however, since the style is not uncommon in extant code, and since a conforming translator is not required to diagnose such mismatches when they occur in separate translation units. Such trouble spots can be made manifest (assuming an implementation provides reasonable diagnostics) by providing new-style function declarations in the translation units with the non-matching calls. Programmers who currently rely on being able to omit trailing arguments are advised to recode using the **<stdarg.h>** paradigm.

Function prototypes may be used to define function types as well:

```
typedef  double (*d_binop) (double A, double B);

struct d_funct {
    d_binop        f1;
    int            (*f2)(double, double);
};
```

The structure **d_funct** has two fields, both of which hold pointers to functions taking two double arguments; the function types differ in their return type.

3.5.5 Type names

Empty parentheses within a type name are always taken as meaning *function with unspecified arguments* and never as (unnecessary) parentheses around the elided identifier. This specification avoids an ambiguity by fiat.

3.5.6 Type definitions

A `typedef` may only be redeclared in an inner block with a declaration that explicitly contains a type name. This rule avoids the ambiguity about whether to take the `typedef` as the type name or the candidate for redeclaration.

Some implementations of C have allowed type specifiers to be added to a type defined using `typedef`. Thus

```
typedef short int small ;
unsigned small x ;
```

would give `x` the type `unsigned short int`. The Committee decided that since this interpretation may be difficult to provide in many implementations, and since it defeats much of the utility of `typedef` as a data abstraction mechanism, such type modifications are invalid. This decision is incorporated in the rules of §3.5.2.

A proposed `typeof` operator was rejected on the grounds of insufficient utility.

3.5.7 Initialization

An implementation might conceivably have codes for floating zero and/or null pointer other than all bits zero. In such a case, the implementation must fill out an incomplete initializer with the various appropriate representations of zero; it may not just fill the area with zero bytes.

The Committee considered proposals for permitting automatic aggregate initializers to consist of a brace-enclosed series of arbitrary (execute-time) expressions, instead of just those usable for a translate-time static initializer. However, cases like this were troubling:

```
int x[2] = { f(x[1]), g(x[0]) };
```

Rather than determine a set of rules which would avoid pathological cases and yet not seem too arbitrary, the Committee elected to permit only static initializers. Consequently, an implementation may choose to build a hidden static aggregate, using the same machinery as for other aggregate initializers, then copy that aggregate to the automatic variable upon block entry.

A structure expression, such as a call to a function returning the appropriate structure type, is permitted as an automatic structure initializer, since the usage seems unproblematic.

For programmer convenience, even though it is a minor irregularity in initializer semantics, the trailing null character in a string literal need not initialize an array element, as in:

```
char mesg[5] = "help!" ;
```

(Some widely used implementations provide precedent.)

The Base Document allows a trailing comma in an initializer at the end of an initializer-list. The Standard has retained this syntax, since it provides flexibility in adding or deleting members from an initializer list, and simplifies machine generation of such lists.

Various implementations have parsed aggregate initializers with partially elided braces differently. The Standard has reaffirmed the (top-down) parse described in the Base Document. Although the construct is allowed, and its parse well defined, the Committee urges programmers to avoid partially elided initializers: such initializations can be quite confusing to read.

QUIET CHANGE

Code which relies on a bottom-up parse of aggregate initializers with partially elided braces will not yield the expected initialized object.

The Committee has adopted the rule (already used successfully in some implementations) that the first member of the union is the candidate for initialization. Other notations for union initialization were considered, but none seemed of sufficient merit to outweigh the lack of prior art.

This rule has a parallel with the initialization of structures. Members of structures are initialized in the sequence in which they are declared. The same can now be said of unions, with the significant difference that only one union member (the first) can be initialized.

3.6   Statements

3.6.1   Labeled statements

Since label definition and label reference are syntactically distinctive contexts, labels are established as a separate name space.

3.6.2   Compound statement, or block

The Committee considered proposals for forbidding a `goto` into a block from outside, since such a restriction would make possible much easier flow optimization and would avoid the whole issue of initializing `auto` storage (see §3.1.2.4). The Committee rejected such a ban out of fear of invalidating working code (however undisciplined) and out of concern for those producing machine-generated C.

3.6.3   Expression and null statements

The `void` cast is not needed in an expression statement, since any value is always discarded. Some checking compilers prefer this reassurance, however, for functions that return objects of types other than `void`.

3.6.4   Selection statements

3.6.4.1   The if statement

See §3.6.2.

3.6.4.2   The switch statement

The controlling expression of a `switch` statement may now have any integral type, even `unsigned long`. Floating types were rejected for switch statements since exact equality in floating point is not portable.

case labels are first converted to the type of the controlling expression of the switch, then checked for equality with other labels; no two may match after conversion.

Case ranges (of the form `lo .. hi`) were seriously considered, but ultimately not adopted in the Standard on the grounds that it added no new capability, just a problematic coding convenience. The construct seems to promise more than it could be mandated to deliver:

- A great deal of code (or jump table space) might be generated for an innocent-looking case range such as `0 .. 65535`.

- The range `'A'..'Z'` would specify all the integers between the character code for `A` and that for `Z`. In some common character sets this range would include non-alphabetic characters, and in others it might not include all the alphabetic characters (especially in non-English character sets).

No serious consideration was given to making the switch more structured, as in Pascal, out of fear of invalidating working code.

QUIET CHANGE

long expressions and constants in switch statements are no longer truncated to `int`.

3.6.5   Iteration statements

3.6.5.1   The while statement

3.6.5.2   The do statement

3.6.5.3   The for statement

3.6.6   Jump statements

3.6.6.1   The goto statement

See §3.6.2.

3.6.6.2   The `continue` statement

The Committee rejected proposed enhancements to `continue` and `break` which would allow specification of an iteration statement other than the immediately enclosing one, on grounds of insufficient prior art.

3.6.6.3   The `break` statement

See §3.6.6.2.

3.6.6.4   The `return` statement

3.7   External definitions

3.7.1   Function definitions

A *function definition* may have its old form (and say nothing about arguments on calls), or it may be introduced by a *prototype* (which affects argument checking and coercion on subsequent calls). (See also §3.1.2.2.)

To avoid a nasty ambiguity, the Standard bans the use of `typedef` names as formal parameters. For instance, in translating the text

```
int f(size_t, a_t, b_t, c_t, d_t, e_t, f_t, g_t,
      h_t, i_t, j_t, k_t, l_t, m_t, n_t, o_t,
      p_t, q_t, r_t, s_t)
```

the translator determines that the construct can only be a prototype declaration as soon as it scans the first `size_t` and following comma. In the absence of this rule, it might be necessary to see the token following the right parenthesis that closes the parameter list, which would require a sizeable look-ahead, before deciding whether the text under scrutiny is a prototype declaration or an old-style function header definition.

An argument list must be explicitly present in the declarator; it cannot be inherited from a `typedef` (see §3.5.4.3). That is to say, given the definition

```
typedef int p(int q, int r);
```

the following fragment is invalid:

```
p funk  /* weird */
{ return q + r ; }
```

Some current implementations rewrite the type of a (for instance) `char` parameter as if it were declared `int`, since the argument is known to be passed as an `int` (in the absence of prototypes). The Standard requires, however, that the received argument be converted *as if* by assignment upon function entry. Type rewriting is thus no longer permissible.

QUIET CHANGE

Functions that depend on `char` or `short` parameter types being widened
to `int`, or `float` to `double`, may behave differently.

Notes for implementors: the assignment conversion for argument passing often
requires no executable code. In most twos-complement machines, a `short` or `char`
is a contiguous subset of the bytes comprising the `int` actually passed (for even
the most unusual byte orderings), so that assignment conversion can be effected by
adjusting the address of the argument (if necessary) .

For an argument declared `float`, however, an explicit conversion must usually
be performed from the `double` actually passed to the `float` desired. Not many
implementations can subset the bytes of a `double` to get a `float`. (Even those that
apparently permit simple truncation often get the wrong answer on certain negative
numbers.)

Some current implementations permit an argument to be masked by a declaration
of the same identifier in the outermost block of a function. This usage is almost
always an erroneous attempt by a novice C programmer to declare the argument;
it is rarely the result of a deliberate attempt to render the argument unreachable.
The Committee decided, therefore, that arguments are effectively declared in the
outermost block, and hence cannot be quietly redeclared in that block.

The Committee considered it important that a function taking a variable number
of arguments, such as `printf`, be expressible portably in C. Hence, the Committee
devoted much time to exploring methods of traversing variable argument lists. One
proposal was to require arguments to be passed as a "brick" (i.e., a contiguous area
of memory), the layout of which would be sufficiently well specified that a portable
method of traversing the brick could be determined.

Several diverse implementations, however, can implement argument passing
more efficiently if the arguments are not required to be contiguous. Thus, the
Committee decided to hide the implementation details of determining the location
of successive elements of an argument list behind a standard set of macros (see §4.8).

3.7.2   External object definitions

See §3.1.2.2.

3.8   Preprocessing directives

For an overview of the philosophy behind the preprocessor, see §2.1.1.2.

Different implementations have had different notions about whether white space
is permissible before and/or after the `#` signalling a preprocessor line. The Com-
mittee decided to allow any white space before the `#`, and horizontal white space

(spaces or tabs) between the # and the directive, since the white space introduces no ambiguity, causes no particular processing problems, and allows maximum flexibility in coding style. Note that similar considerations apply for comments, which are reduced to white space early in the phases of translation (§2.1.1.2):

```
/* here a comment */ #if BLAH
#/* there a comment */ if BLAH
# if /* every-
    where a comment */ BLAH
```

The lines all illustrate legitimate placement of comments.

3.8.1   Conditional inclusion

For a discussion of evaluation of expressions following #if, see §3.4.

The operator defined has been added to make possible writing boolean combinations of defined flags with one another and with other inclusion conditions. If the identifier defined were to be defined as a macro, defined(X) would mean the macro expansion in C text proper and the operator expression in a preprocessing directive (or else that the operator would no longer be available). To avoid this problem, such a definition is not permitted (§3.8.8).

#elif has been added to minimize the stacking of #endif directives in multi-way conditionals.

Processing of skipped material is defined such that an implementation need only examine a logical line for the # and then for a directive name. Thus, assuming that xxx is undefined, in this example:

```
# ifndef xxx
# define xxx "abc"
# elif xxx > 0
    /* ... */
# endif
```

an implementation is not required to diagnose an error for the elif statement, even though if it *were* processed, a syntactic error would be detected.

Various proposals were considered for permitting text other than comments at the end of directives, particularly #endif and #else, presumably to label them for easier matchup with their corresponding #if directives. The Committee rejected all such proposals because of the difficulty of specifying exactly what would be permitted, and how the translator would have to process it.

Various proposals were considered for permitting additional unary expressions to be used for the purpose of testing for the system type, testing for the presence of a file before #include, and other extensions to the preprocessing language. These proposals were all rejected on the grounds of insufficient prior art and/or insufficient utility.

3.8.2 Source file inclusion

Specification of the `#include` directive raises distinctive grammatical problems because the file name is conventionally parsed quite differently than an "ordinary" token sequence:

- The angle brackets are not operators, but delimiters.

- The double quotes do not delimit a string literal with all its defined escape sequences. (In some systems, backslash is a legitimate character in a filename.) The construct just looks like a string literal.

- White space or characters not in the C repertoire may be permissible and significant within either or both forms.

These points in the description of phases of translation are of particular relevance to the parse of the `#include` directive:

- Any character otherwise unrecognized during tokenization is an instance of an "invalid token." As with valid tokens, the spelling is retained so that later phases can, if necessary, map a token sequence (back) into a sequence of characters.

- Preprocessing phases must maintain the spelling of preprocessing tokens; the filename is based on the original spelling of the tokens, not on any interpretation of escape sequences.

- The filename on the `#include` (and `#line`) directive, if it does not begin with " or <, is macro expanded prior to execution of the directive. Allowing macros in the `include` directive facilitates the parameterization of include file names, an important issue in transportability.

The file search rules used for the filename in the `#include` directive were left as implementation-defined. The Standard intends that the rules which are eventually provided by the implementor correspond as closely as possible to the original K&R rules. The primary reason that explicit rules were not included in the Standard is the infeasibility of describing a portable file system structure. It was considered unacceptable to include UNIX-like directory rules due to significant differences between this structure and other popular commercial file system structures.

Nested include files raise an issue of interpreting the file search rules. In UNIX C an include statement found within an include file entails a search for the named file relative to the file system *directory* that holds the outer `#include`. Other implementations, including the earlier UNIX C described in K&R, always search relative to the same *current directory*. The Committee decided, in principle, in favor of the K&R approach, but was unable to provide explicit search rules as explained above.

The Standard specifies a set of include file names which must map onto distinct host file names. In the absence of such a requirement, it would be impossible to write portable programs using include files.

Section §2.2.4.1 on translation limits contains the required number of nesting levels for include files. The limits chosen were intended to reflect reasonable needs for users constrained by reasonable system resources available to implementors.

By defining a failure to read an include file as a syntax error, the Standard requires that the failure be diagnosed. More than one proposal was presented for some form of conditional include, or a directive such as `#ifincludable`, but none were accepted by the Committee due to lack of prior art.

3.8.3   Macro replacement

The specification of macro definition and replacement in the Standard was based on these principles:

- Interfere with existing code as little as possible.

- Keep the preprocessing model simple and uniform.

- Allow macros to be used wherever functions can be.

- Define macro expansion such that it produces the same token sequence whether the macro calls appear in open text, in macro arguments, or in macro definitions.

Preprocessing is specified in such a way that it can be implemented as a separate (text-to-text) pre-pass or as a (token-oriented) portion of the compiler itself. Thus, the preprocessing grammar is specified in terms of tokens.

However, the new-line character must be a token during preprocessing, because the preprocessing grammar is line-oriented. The presence or absence of white space is also important in several contexts, such as between the macro name and a following parenthesis in a `#define` directive. To avoid overly constraining the implementation, the Standard allows the preservation of each white space character (which is easy for a text-to-text pre-pass) or the mapping of white space into a single "white space" token (which is easier for token-oriented translators).

The Committee desired to disallow "pernicious redefinitions" such as
(in header1.h)

        #define NBUFS 10

(in header2.h)

        #define NBUFS 12

which are clearly invitations to serious bugs in a program. There remained, however, the question of "benign redefinitions," such as

(in header1.h)

```
#define NULL_DEV 0
```

(in header2.h)

```
#define NULL_DEV 0
```

The Committee concluded that safe programming practice is better served by allowing benign redefinition where the definitions are the same. This allows independent headers to specify their understanding of the proper value for a symbol of interest to each, with diagnostics generated only if the definitions differ.

The definitions are considered "the same" if the identifier-lists, token sequences, and occurrences of white-space (ignoring the spelling of white-space) in the two definitions are identical.

Existing implementations have differed on whether keywords can be redefined by macro definitions. The Committee has decided to allow this usage; it sees such redefinition as useful during the transition from existing to Standard-conforming translators.

These definitions illustrate possible uses:

```
# define char    signed char
# define sizeof (int) sizeof
# define const
```

The first case might be useful in moving extant code from a signed-char implementation to one in which `char` is unsigned. The second case might be useful in adapting code which assumes that `sizeof` results in an `int` value. The redefinition of `const` could be useful in retrofitting more modern C code to an older implementation.

As with any other powerful language feature, keyword redefinition is subject to abuse. Users cannot expect any meaningful behavior to come about from source files starting with

```
#define int double
#include <stdio.h>
```

or similar subversions of common sense.

3.8.3.1   Argument substitution

3.8.3.2   The # operator

Some implementations have decided to replace identifiers found within a string literal if they match a macro argument name. The replacement text is a "stringized" form of the actual argument token sequence. This practice appears to be contrary to the definition, in K&R, of preprocessing in terms of token sequences. The Committee declined to elaborate the syntax of string literals to the point where this

practice could be condoned. However, since the facility provided by this mechanism seems to be widely used, the Committee introduced a more tractable mechanism of comparable power.

The `#` operator has been introduced for stringizing. It may only be used in a `#define` expansion. It causes the formal parameter name following to be replaced by a string literal formed by stringizing the actual argument token sequence. In conjunction with string literal concatenation (see §3.1.4), use of this operator permits the construction of strings as effectively as by identifier replacement within a string. An example in the Standard illustrates this feature.

One problem with defining the effect of stringizing is the treatment of white space occurring in macro definitions. Where this could be discarded in the past, now upwards of one logical line worth (over 500 characters) may have to be retained. As a compromise between token-based and character-based preprocessing disciplines, the Committee decided to permit white space to be retained as one bit of information: none or one. Arbitrary white space is replaced in the string by one space character.

The remaining problem with stringizing was to associate a "spelling" with each token. (The problem arises in token-based preprocessors, which might, for instance, convert a numeric literal to a canonical or internal representation, losing information about base, leading 0's, etc.) In the interest of simplicity, the Committee decided that each token should expand to just those characters used to specify it in the original source text.

QUIET CHANGE

A macro that relies on formal parameter substitution within a string literal will produce different results.

3.8.3.3   The `##` operator

Another facility relied on in much current practice but not specified in the Base Document is "token pasting," or building a new token by macro argument substitution. One existing implementation is to replace a comment within a macro expansion by zero characters, instead of the single space called for in K&R. The Committee considered this practice unacceptable.

As with "stringizing," the facility was considered desirable, but not the extant implementation of this facility, so the Committee invented another preprocessing operator. The `##` operator within a macro expansion causes concatenation of the tokens on either side of it into a new composite token. The specification of this pasting operator is based on these principles:

- Paste operations are explicit in the source.

- The `##` operator is associative.

- A formal parameter as an operand for `##` is not expanded before pasting. (The actual is substituted for the formal, but the actual is not expanded:

```
#define a(n) aaa ## n
#define b    2
```

Given these definitions, the expansion of `a(b)` is `aaab`, not `aaa2` or `aaan`.)

- A normal operand for `##` is not expanded before pasting.

- Pasting does not cross macro replacement boundaries.

- The token resulting from a paste operation is subject to further macro expansion.

These principles codify the essential features of prior art, and are consistent with the specification of the stringizing operator.

3.8.3.4 Rescanning and further replacement

A problem faced by most current preprocessors is how to use a macro name in its expansion without suffering "recursive death." The Committee agreed simply to turn off the definition of a macro for the duration of the expansion of that macro. An example of this feature is included in the Standard.

The rescanning rules incorporate an ambiguity. Given the definitions

```
#define  f(a)  a*g
#define  g     f
```

it is clear (or at least unambiguous) that the expansion of `f(2)(9)` is `2*f(9)` — the `f` in the result clearly was introduced during the expansion of the original `f`, so is not further expanded.

However, given the definitions

```
#define f(a)  a*g
#define g(a)  f(a)
```

the expansion rules allow the result to be either `2*f(9)` or `2*9*g` — it is unclear whether the `f(9)` token string (resulting from the initial expansion of `f` and the examination of the rest of the source file) should be considered as nested within the expansion of `f` or not. The Committee intentionally left this behavior ambiguous: it saw no useful purpose in specifying all the quirks of preprocessing for such questionably useful constructs.

3.8.3.5 Scope of macro definitions

Some pre-Standard implementations maintain a stack of `#define` instances for each identifier; `#undef` simply pops the stack. The Committee agreed that more than one level of `#define` was more prone to error than utility.

It is explicitly permitted to `#undef` a macro that has no current definition. This capability is exploited in conjunction with the standard library (see §4.1.3).

3.8.4   Line control

Aside from giving values to __LINE__ and __FILE__ (see §3.8.8), the effect of #line is unspecified. A good implementation will presumably provide line and file information in conjunction with most diagnostics.

3.8.5   Error directive

The directive #error has been introduced to provide an explicit mechanism for forcing translation to fail under certain conditions. (Formally the Standard only requires, *can* only require, that a diagnostic be issued when the #error directive is effected. It is the intent of the Committee, however, that translation cease immediately upon encountering this directive, if this is feasible in the implementation; further diagnostics on text beyond the directive are apt to be of little value.) Traditionally such failure has had to be forced by inserting text so ill-formed that the translator gagged on it.

3.8.6   Pragma directive

The #pragma directive has been added as the universal method for extending the space of directives.

3.8.7   Null directive

The existing practice of using empty # lines for spacing is supported in the Standard.

3.8.8   Predefined macro names

The rule that these macros may not be redefined or undefined reduces the complexity of the name space that the programmer and implementor must understand; it recognizes that these macros have special built-in properties.

   The macros __DATE__ and __TIME__ have been added to make available the time of translation. A particular format for the expansion of these macros has been specified to aid in parsing strings initialized by them.

   The macros __LINE__ and __FILE__ have been added to give programmers access to the source line number and file name.

   The macro __STDC__ allows for conditional translation on whether the translator claims to be standard-conforming or not. It is defined as having value 1; future versions of the Standard could define it as 2, 3, ..., to allow for conditional compilation on which version of the Standard a translator conforms to. This macro should be of use in the transition toward conformance to the Standard.

3.9   Future language directions

This section includes specific mention of the future direction in which the Committee intends to extend and/or restrict the language. The contents of this section should be considered as quite likely to become a part of the next version of the Standard. Implementors are advised that failure to take heed of the points mentioned herein is considered undesirable for a conforming hosted or freestanding implementation. Users are advised that failure to take heed of the points mentioned herein is considered undesirable for a conforming program.

3.9.1   External names

3.9.2   Character escape sequences

3.9.3   Storage-class specifiers

See §3.5.1.

3.9.4   Function declarators

The characterization as obsolescent of the use of the "old style" function declarations and definitions — that is, the traditional style not using prototypes — signals the Committee's intent that the new prototype style should eventually replace the old style.

The case for the prototype style is presented in §3.3.2.2 and §3.5.4.3. The gist of this case is that the new syntax addresses some of the most glaring weaknesses of the language defined in the Base Document, that the new style is superior to the old style on every count.

It was obviously out of the question to remove syntax used in the overwhelming majority of extant C code, so the Standard specifies two ways of writing function declarations and function definitions. Characterizing the old style as obsolescent is meant to discourage its use, and to serve as a strong endorsement by the Committee of the new style. It confidently expects that approval and adoption of the prototype style will make it feasible for some future C Standard to remove the old style syntax.

3.9.5   Function definitions

See §3.9.4.

3.9.6   Array parameters

As vector and parallel hardware, and numeric applications in C, become more common, the aliasing semantics of C have been a source of frustration for implementors wanting to make optimum use of such hardware. If arrays are known not to overlap, certain optimizations become possible, but C currently provides no way to specify to a translator that argument arrays indeed do not overlap. The Committee, in

adopting this future direction, hopes to provide common ground for implementors and users concerned with this problem, so that some future C Standard can adopt this non-overlapping rule on the basis of widespread experience.

Section 4

LIBRARY

4.1   Introduction

The Base Document for this section of the Standard was the *1984 /usr/group Standard*. The /usr/group document contains definitions of some facilities which were specific to the UNIX Operating System and not relevant to other operating environments, such as pipes, ioctls, file access permissions and process control facilities. Those definitions were dropped from the Standard. Some other functions were excluded from the Standard because they were non-portable or were ill-defined.

Other facilities not in the library Base Document but present in many UNIX implementations, such as the curses (terminal-independent screen handling) library were considered to be more complex and less essential than the facilities of the Base Document; these functions were not added to the Standard.

4.1.1   Definitions of terms

The *decimal-point character* is the character used in the input or output of floating point numbers, and may be changed by `setlocale`. This is a library construct; the decimal point in numeric literals in C source text is always a period.

4.1.2   Standard headers

Whereas in prior practice only certain library functions have been associated with header files, the Standard now mandates that *all* library functions have a header. Several headers have therefore been added, and the contents of a few old ones have been changed.

In many implementations the names of headers are the names of files in special directories. This implementation technique is not required, however: the Standard makes no assumptions about the form that a file name may take on any system. Headers may thus have a special status if an implementation so chooses. Standard headers may even be built into a translator, provided that their contents do not become "known" until after they are explicitly included. One purpose of permitting

71

these header "files" to be "built in" to the translator is to allow an implementation of the C language as an interpreter in an un-hosted environment, where the only "file" support may be a network interface.

The Committee decided to make library headers "idempotent" — they should be includable any number of times, and includable in any order. This requirement, which reflects widespread existing practice, may necessitate some protective wrappers within the headers, to avoid, for instance, redefinitions of typedefs. To ensure that such protective wrapping can be made to work, and to ensure proper scoping of typedefs, headers may only be included outside of any declaration.

Note to implementors: a common way of providing this "protective wrapping" is:

```
#ifndef __ERRNO_H
#define __ERRNO_H
/* body of <errno.h> */
/* ...                 */
#endif
```

where __ERRNO_H is an otherwise unused macro name.

Implementors often desire to provide implementations of C in addition to that pre-scribed by the Standard. For instance, an implementation may want to provide system-specific I/O facilities in <stdio.h>. A technique that allows the same header to be used in both the Standard-conforming and alternate implementations is to add the extra, non-Standard, declarations to the header as in this illustration:

```
#ifdef __EXTENSIONS__
typedef int file_no;
extern int read(file_no _N, void * _Buffer, int _Nbytes);
/*...*/
#endif
```

The header is usable in an implementation of the Standard in the absence of a definition of __EXTENSIONS__, and the non-Standard implementation can provide the appropriate definitions to enable the extra declarations.

4.1.2.1   Reserved identifiers

To give implementors maximum latitude in packing library functions into files, all external identifiers defined by the library are reserved (in a hosted environment). This means, in effect, that no user supplied external names may match library names, *not even if the user function has the same specification*. Thus, for instance, strtod may be defined in the same object module as printf, with no fear that link-time conflicts will occur. Equally, strtod may call printf, or printf may call strtod, for whatever reason, with no fear that the wrong function will be called.

Also reserved for the implementor are *all* external identifiers beginning with an underscore, and all other identifiers beginning with an underscore followed by a capital letter or an underscore. This gives a space of names for writing the numerous behind-the-scenes non-external macros and functions a library needs to do its job properly.

With these exceptions, the Standard assures the programmer that *all other* identifiers are available, with no fear of unexpected collisions when moving programs from one implementation to another.[1] Note, in particular, that part of the name space of internal identifiers beginning with underscore is available to the user — translator implementors have not been the only ones to find use for "hidden" names. C is such a portable language in many respects that this issue of "name space pollution" is currently one of the principal barriers to writing completely portable code. Therefore the Standard assures that macro and typedef names are reserved only if the associated header is explicitly included.

4.1.3 Errors
`<errno.h>`

`<errno.h>` is a header invented to encapsulate the error handling mechanism used by many of the library routines in `math.h` and `strlib.h`.[2]

The error reporting machinery centered about the setting of `errno` is generally regarded with tolerance at best. It requires a "pathological coupling" between library functions and makes use of a static writable memory cell, which interferes with the construction of shareable libraries. Nevertheless, the Committee preferred to standardize this existing, however deficient, machinery rather than invent something more ambitious.

The definition of `errno` as an lvalue macro grants implementors the license to expand it to something like `*__errno_addr()`, where the function returns a pointer to the (current) modifiable copy of `errno`.

4.1.4 Limits
`<float.h>` and `<limits.h>`

Both `<float.h>` and `<limits.h>` are inventions. Included in these headers are various parameters of the execution environment which are potentially useful at compile time, and which are difficult or impossible to determine by other means.

The availability of this information in headers provides a portable way of tuning a program to different environments. Another possible method of determining

---

[1]See §3.1.2.1 for a discussion of some of the precautions an implementor should take to keep this promise. Note also that any implementation-defined member names in structures defined in `<time.h>` and `<locals.h>` must begin with an underscore, rather than following the pattern of other names in those structures.

[2]In earlier drafts of the Standard, errno and related macros were defined in `<stddef.h>`. When the Committee decided that the other definitions in this header were of such general utility that they should be required even in freestanding environments, it created `<errno.h>`.

some of this information is to evaluate arithmetic expressions in the preprocessing statements. Requiring that preprocessing always yield the same results as run-time arithmetic, however, would cause problems for portable compilers (themselves written in C) or for cross compilers, which would then be required to implement the (possibly wildly different) arithmetic of the target machine on the host machine. (See §3.4.)

`<float.h>` makes available to programmers a set of useful quantities for numerical analysis. (See §2.2.4.2.) This set of quantities has seen widespread use for such analysis, in C and in other languages, and was recommended by the numerical analysts on the Committee. The set was chosen so as not to prejudice an implementation's selection of floating-point representation.

Most of the limits in `<float.h>` are specified to be general `double` expressions rather than restricted constant expressions

- to allow use of values which cannot readily (or, in some cases, cannot possibly) be constructed as manifest constants, and

- to allow for run-time selection of floating-point properties, as is possible, for instance, in IEEE-854 implementations.

4.1.5   Common definitions
`<stddef.h>`

`<stddef.h>` is a header invented to provide definitions of several types and macros used widely in conjunction with the library: `ptrdiff_t` (see §3.3.6), `size_t` (see §3.3.3.4), `wchar_t` (see §3.1.3.4), and `NULL`. Including any header that references one of these macros will also define it, an exception to the usual library rule that each macro or function belongs to exactly one header.

`NULL` can be defined as any *null pointer constant*. Thus existing code can retain definitions of `NULL` as `0` or `0L`, but an implementation may choose to define it as `(void *)0`; this latter form of definition is convenient on architectures where the pointer size(s) do(es) not equal the size of any integer type. It has never been wise to use `NULL` in place of an arbitrary pointer as a function argument, however, since pointers to different types need not be the same size. The library avoids this problem by providing special macros for the arguments to `signal`, the one library function that might see a null function pointer.

The `offsetof` macro has been added to provide a portable means of determining the offset, in bytes, of a member within its structure. This capability is useful in programs, such as are typical in data-base implementations, which declare a large number of different data structures: it is desirable to provide "generic" routines that work from descriptions of the structures, rather than from the structure declarations themselves.[3]

---

[3] Consider, for instance, a set of nodes (structures) which are to be dynamically allocated and

In many implementations, `offsetof` could be defined as one of

```
(size_t)&(((s_name*)0)->m_name)
```

or

```
(size_t)(char *)&(((s_name*)0)->m_name)
```

or, where X is some predeclared address (or 0) and `A(Z)` is defined as `((char*)&Z)`,

```
(size_t)( A( (s_name*)X->m_name ) - A( X ))
```

It was not feasible, however, to mandate any single one of these forms as a construct guaranteed to be portable.

Other implementations may choose to expand this macro as a call to a built-in function that interrogates the translator's symbol table.

4.1.6 Use of library functions

To make usage more uniform for both implementor and programmer, the Standard requires that every library function (unless specifically noted otherwise) must be represented as an actual function, in case a program wishes to pass its address as a parameter to another function. On the other hand, every library function is now a candidate for redefinition, in its associated header, as a macro, provided that the macro performs a "safe" evaluation of its arguments, i.e., it evaluates each of the arguments exactly once and parenthesizes them thoroughly, and provided that its top-level operator is such that the execution of the macro is not interleaved with other expressions. Two exceptions are the macros `getc` and `putc`, which may evaluate their arguments in an unsafe manner. (See §4.9.7.5.)

If a program requires that a library facility be implemented as an actual function, not as a macro, then the macro name, if any, may be erased by using the `#undef` preprocessing directive (see §3.8.3).

All library prototypes are specified in terms of the "widened" types: an argument formerly declared as `char` is now written as `int`. This ensures that most library functions can be called with or without a prototype in scope (see §3.3.2.2), thus maintaining backwards compatibility with existing, pre-Standard, code. Note, however, that since functions like `printf` and `scanf` use variable-length argument lists, they must be called in the scope of a prototype.

The Standard contains an example showing how certain library functions may be "built in" in an implementation that remains *conforming*.

---

garbage-collected, and which can contain pointers to other such nodes. A possible implementation is to have the first field in each node point to a descriptor for that node. The descriptor includes a table of the offsets of fields which are pointers to other nodes. A garbage-collector "mark" routine needs no further information about the content of the node (except, of course, where to put the mark). New node types can be added to the program without requiring the mark routine to be rewritten or even recompiled.

4.2   Diagnostics
`<assert.h>`

4.2.1   Program diagnostics

4.2.1.1   The assert macro

Some implementations tolerate an arbitrary scalar expression as the argument to
assert, but the Committee decided to require correct operation only for int ex-
pressions. For the sake of implementors, no hard and fast format for the output
of a failing assertion is required; but the Standard mandates enough machinery to
replicate the form shown in the footnote.

It can be difficult or impossible to make assert a true function, so it is restricted
to macro form only.

To minimize the number of different methods for program termination, assert
is now defined in terms of the abort function.

Note that defining the macro NDEBUG to disable assertions may change the be-
havior of a program with no failing assertion if any argument expression to assert
has side-effects, because the expression is no longer evaluated.

It is possible to turn assertions off and on in different functions within a transla-
tion unit by defining (or undefining) NDEBUG and including <assert.h> again. The
implementation of this behavior in <assert.h> is simple: undefine any previous
definition of assert before providing the new one. Thus the header might look like

```
#undef assert
#ifdef NDEBUG
 #define assert(ignore)  ((void) 0)
#else
 extern void __gripe(char *_Expr, char *_File, int _Line);
 #define assert(expr) \
    ( (expr)? (void)0 : __gripe(#expr, __FILE__, __LINE__) )
#endif
```

Note that assert must expand to a void expression, so the more obvious if state-
ment does not suffice as a definition of assert. Note also the avoidance of names
in a header which would conflict with the user's name space (see §3.1.2.1).

4.3   Character Handling
`<ctype.h>`

Pains were taken to eliminate any ASCII dependencies from the definition of the
character handling functions. One notable result of this policy was the elimination
of the function isascii, both because of the name and because its function was hard
to generalize. Nevertheless, the character functions are often most clearly explained
in concrete terms, so ASCII is used frequently to express examples.

Since these functions are often used primarily as macros, their domain is restricted to the small positive integers representable in an **unsigned char**, plus the value of **EOF**. **EOF** is traditionally −1, but may be any negative integer, and hence distinguishable from any valid character code. These macros may thus be efficiently implemented by using the argument as an index into a small array of attributes.

The Standard (§4.13.1) warns that names beginning with **is** and **to**, when these are followed by lower-case letters, are subject to future use in adding items to `<ctype.h>`.

4.3.1 Character testing functions

The definitions of *printing character* and *control character* have been generalized from ASCII.

Note that none of these functions returns a nonzero value (true) for the argument value **EOF**.

4.3.1.1 The `isalnum` function

4.3.1.2 The `isalpha` function

The Standard specifies that the set of letters, in the default *locale*, comprises the 26 upper-case and 26 lower-case letters of the Latin (English) alphabet. This set may vary in a *locale-specific* fashion (that is, under control of the **setlocale** function, §4.4) so long as

- `isupper(c)` implies `isalpha(c)`

- `islower(c)` implies `isalpha(c)`

- `isspace(c)`, `ispunct(c)`, `iscntrl(c)`, or `isdigit(c)` implies `!isalpha(c)`

4.3.1.3 The `iscntrl` function

4.3.1.4 The `isdigit` function

4.3.1.5 The `isgraph` function

4.3.1.6 The `islower` function

4.3.1.7 The `isprint` function

4.3.1.8 The `ispunct` function

4.3.1.9 The `isspace` function

`isspace` is widely used within the library as the working definition of white space.

4.3.1.10   The `isupper` function

4.3.1.11   The `isxdigit` function

4.3.2   Character case mapping functions

Earlier libraries had (almost equivalent) macros, `_tolower` and `_toupper`, for these functions. The Standard now permits any library function to be additionally implemented as a macro; the underlying function must still be present. `_toupper` and `_tolower` are thus unnecessary and were dropped as part of the general standardization of library macros.

4.3.2.1   The `tolower` function

4.3.2.2   The `toupper` function

4.4   Localization
`<locale.h>`

C has become an international language. Users of the language outside the United States have been forced to deal with the various Americanisms built into the standard library routines.

Areas affected by international considerations include:

Alphabet. The English language uses 26 letters derived from the Latin alphabet. This set of letters suffices for English, Swahili, and Hawaiian; all other living languages use either the Latin alphabet *plus* other characters, or other, non-Latin alphabets or syllabaries.

In English, each letter has an upper-case and lower-case form. The German "sharp S", ß, occurs only in lower-case. European French usually omits diacriticals on upper-case letters. Some languages do not have the concept of two cases.

Collation. In both EBCDIC and ASCII the code for 'z' is greater than the code for 'a', and so on for other letters in the alphabet, so a "machine sort" gives not unreasonable results for ordering strings. In contrast, most European languages use a codeset resembling ASCII in which some of the codes used in ASCII for punctuation characters are used for alphabetic characters. (See §2.2.1.) The ordering of these codes is not alphabetic. In some languages letters with diacritics sort as separate letters; in others they should be collated just as the unmarked form. In Spanish, "ll" sorts as a single letter following "l"; in German, "ß" sorts like "ss".

Formatting of numbers and currency amounts. In the United States the period is invariably used for the decimal point; this usage was built into the definitions of such functions as `printf` and `scanf`. Prevalent practice in several major European countries is to use a comma; a raised dot is employed

in some locales. Similarly, in the United States a comma is used to separate groups of three digits to the left of the decimal point; a period is common in Europe, and in some countries digits are not grouped by threes. In printing currency amounts, the currency symbol (which may be more than one character) may precede, follow, or be embedded in the digits.

Date and time. The standard function `asctime` returns a string which includes abbreviations for month and weekday names, and returns the various elements in a format which might be considered unusual even in its country of origin.

Various common date formats include

1776-07-04	ISO Format
4.7.76	customary central European and British usage
7/4/76	customary U.S. usage
4.VII.76	Italian usage
76186	Julian date (YYDDD)
04JUL76	airline usage
Thursday, July 4, 1776	full U.S. format
Donnerstag, 4. Juli 1776	full German format

Time formats are also quite diverse:

3:30 PM	customary U.S. and British format
1530	U.S. military format
15h.30	Italian usage
15.30	German usage
15:30	common European usage

The Committee has introduced mechanisms into the C library to allow these and other issues to be treated in the appropriate *locale-specific* manner.

The localization features of the Standard are based on these principles:

English for C source. The C language proper is based on English. Keywords are based on English words. A program which uses "national characters" in identifiers is not strictly conforming. (Use of national characters in comments is strictly conforming, though what happens when such a program is printed in a different locale is unspecified.) The decimal point must be a period in C source, and no thousands delimiter may be used.

Runtime selectability. The locale must be selectable at runtime, from an implementation-defined set of possibilities. Translate-time selection does not offer sufficient flexibility. Software vendors do not want to supply different

object forms of their programs in different locales. Users do not want to use different versions of a program just because they deal with several different locales.

Function interface. Locale is changed by calling a function, thus allowing the implementation to recognize the change, rather than by, say, changing a memory location that contains the decimal point character.

Immediate effect. When a new locale is selected, affected functions reflect the change immediately. (This is not meant to imply if a signal-handling function were to change the selected locale and return to a library function, that the return value from that library function must be completely correct with respect to the new locale.)

4.4.1   Locale control

4.4.1.1   The `setlocale` function

`setlocale` provides the mechanism for controlling *locale-specific* features of the library. The `category` argument allows parts of the library to be localized as necessary without changing the entire locale-specific environment. Specifying the `locale` argument as a string gives an implementation maximum flexibility in providing a set of locales. For instance, an implementation could map the argument string into the name of a file containing appropriate localization parameters — these files could then be added and modified without requiring any recompilation of a localizable program.

4.4.2   Numeric formatting convention inquiry

4.4.2.1   The `localeconv` function

The `localeconv` function gives a programmer access to information about how to format numeric quantities (monetary or otherwise). This sort of interface was considered preferable to defining conversion functions directly: even with a specified locale, the set of distinct formats that can be constructed from these elements is large, and the ones desired very application-dependent.

4.5   Mathematics
`<math.h>`

For historical reasons, the math library is only defined for the floating type `double`. All the names formed by appending `f` or `l` to a name in `<math.h>` are reserved to allow for the definition of `float` and `long double` libraries.

The functions `ecvt`, `fcvt`, and `gcvt` have been dropped since their capability is available through `sprintf`.

Traditionally, `HUGE_VAL` has been defined as a manifest constant that approximates the largest representable `double` value. As an approximation to *infinity* it is problematic. As a function return value indicating overflow, it can cause trouble if first assigned to a `float` before testing, since a `float` may not necessarily hold all values representable in a `double`.

After considering several alternatives, the Committee decided to generalize `HUGE_VAL` to a positive double expression, so that it could be expressed as an external identifier naming a location initialized precisely with the proper bit pattern. It can even be a special encoding for *machine infinity*, on implementations that support such codes. It need not be representable as a `float`, however.

Similarly, domain errors in the past were typically indicated by a zero return, which is not necessarily distinguishable from a valid result. The Committee agreed to make the return value for domain errors *implementation-defined*, so that special machine codes can be used to advantage. This makes possible an implementation of the math library in accordance with the IEEE P854 proposal on floating point representation and arithmetic.

4.5.1 Treatment of error conditions

Whether underflow should be considered a range error, and cause `errno` to be set, is specified as *implementation-defined* since detection of underflow is inefficient on some systems.

The Standard has been crafted to neither require nor preclude any popular implementation of floating point. This principle affects the definition of *domain error*: an implementation may define extra domain errors to deal with floating-point arguments such as infinity or "not-a-number".

The Committee considered the adoption of the `matherr` capability from UNIX System V. In this feature of that system's math library, any error (such as overflow or underflow) results in a call from the library function to a user-defined exception handler named `matherr`. The Committee rejected this approach for several reasons:

- This style is incompatible with popular floating point implementations, such as IEEE 754 (with its special return codes), or that of VAX/VMS.

- It conflicts with the error-handling style of FORTRAN, thus making it more difficult to translate useful bodies of mathematical code from that language to C.

- It requires the math library to be reentrant (since math routines could be called from `matherr`), which may complicate some implementations.

- It introduces a new style of library interface: a user-defined library function with a library-defined name. Note, by way of comparison, the signal and exit handling mechanisms, which provide a way of "registering" user-defined functions.

4.5.2   Trigonometric functions

Implementation note: trignometric argument reduction should be performed by a method that causes no catastrophic discontinuities in the error of the computed result. In particular, methods based solely on naive application of a calculation like

```
x - (2*pi) * (int)(x/(2*pi))
```

are ill-advised.

4.5.2.1   The acos function

4.5.2.2   The asin function

4.5.2.3   The atan function

4.5.2.4   The atan2 function

The `atan2` function is modelled after FORTRAN's. It is described in terms of $\arctan \frac{y}{x}$ for simplicity; the Committee did not wish to complicate the descriptions by specifying in detail how the determine the appropriate quadrant, since that should be obvious from normal mathematical convention. `atan2(y,x)` is well-defined and finite, even when `x` is 0; the one ambiguity occurs when both arguments are 0, because at that point any value in the range of the function could logically be selected. Since valid reasons can be advanced for all the different choices that have been in this situation by various implements, the Standard preserves the implementor's freedom to return an arbitrary well-defined value such as 0, to report a domain error, or to return an IEEE *NaN* code.

4.5.2.5   The cos function

4.5.2.6   The sin function

4.5.2.7   The tan function

The tangent function has singularities at odd multiples of $\frac{\pi}{2}$, approaching $+\infty$ from one side and $-\infty$ from the other. Implementations commonly perform argument reduction using the best machine representation of π; for arguments to `tan` sufficiently close to a singularity, such reduction may yield a value on the wrong side of the singularity. In view of such problems, the Committee has recognized that `tan` is an exception to the *range error* rule (§4.5.1) that an overflowing result produces `HUGE_VAL` properly signed.)

4.5.3 Hyperbolic functions

4.5.3.1 The cosh function

4.5.3.2 The sinh function

4.5.3.3 The tanh function

4.5.4 Exponential and logarithmic functions

4.5.4.1 The exp function

4.5.4.2 The frexp function

The functions `frexp`, `ldexp`, and `modf` are primitives used by the remainder of the library. There was some sentiment for dropping them for the same reasons that `ecvt`, `fcvt`, and `gcvt` were dropped, but their adherents rescued them for general use. Their use is problematic: on nonbinary architectures `ldexp` may lose precision, and `frexp` may be inefficient.

4.5.4.3 The ldexp function

See §4.5.4.2.

4.5.4.4 The log function

Whether `log(0.)` is a domain error or a range error is arguable. The choice in the Standard, *range error*, is for compatibility with IEEE P854. Some such implementations would represent the result as $-\infty$, in which case no error is raised.

4.5.4.5 The log10 function

See §4.5.4.4.

4.5.4.6 The modf function

See §4.5.4.2.

4.5.5 Power functions

4.5.5.1 The pow function

4.5.5.2 The sqrt function

IEEE P854, unlike the Standard, requires `sqrt(-0.)` to return a negatively signed magnitude-zero result. This is an issue on implementations that support a negative floating zero. The Standard specifies that taking the square root of a negative number (in the mathematical sense: less than 0) is a domain error which requires the function to return an *implementation-defined* value. This rule permits

implementations to support either the IEEE P854 or vendor-specific floating point representations.

4.5.6   Nearest integer, absolute value, and remainder functions

4.5.6.1   The `ceil` function

Implementation note: The `ceil` function returns the smallest integral value in double format not less than x, even though that integer might not be representable in a C integral type. `ceil(x)` equals x for all x sufficiently large in magnitude. An implementation that calculates `ceil(x)` as

```
(double)(int) x
```

is ill-advised.

4.5.6.2   The `fabs` function

Adding an absolute value operator was rejected by the Committee. An implementation can provide a built-in function for efficiency.

4.5.6.3   The `floor` function

4.5.6.4   The `fmod` function

`fmod` is defined even if the quotient x/y is not representable — this function is properly implemented by scaled subtraction rather than by division. The Standard defines the result in terms of the formula $x - i*y$, where i is some integer. This integer need not be representable, and need not even be explicitly computed. Thus implementations are advised not to compute the result using a formula like

```
x - y * (int)(x/y)
```

Instead, the result can be computed in principle by subtracting `ldexp(y,n)` from x, for appropriately chosen decreasing n, until the remainder is between 0 and x — efficiency considerations may dictate a different actual implementation.

   The result of `fmod(x,0.0)` is either a domain error or 0.0; the result always lies between 0.0 and y, so specifying the non-erroneous result as 0.0 simply recognizes the limit case.

   The Committee considered and rejected a proposal to use the remainder operator % for this function; the operators in general correspond to hardware facilities, and `fmod` is not supported in hardware on most machines.

4.6   Nonlocal jumps
   `<setjmp.h>`

`jmp_buf` must be an array type for compatibility with existing practice: programs typically omit the address operator before a `jmp_buf` argument, even though a

pointer to the argument is desired, not the value of the argument itself. Thus, a scalar or struct type is unsuitable. Note that a one-element array of the appropriate type is a valid definition.

`setjmp` is constrained to be a macro only: in some implementations the information necessary to restore context is only available while executing the function making the call to `setjmp`.

4.6.1   Save calling environment

4.6.1.1   The `setjmp` macro

One proposed requirement on `setjmp` is that it be usable like any other function — that it be callable in any expression context, and that the expression evaluate correctly whether the return from `setjmp` is direct or via a call to `longjmp`. Unfortunately, any implementation of `setjmp` as a conventional called function cannot know enough about the calling environment to save any temporary registers or dynamic stack locations used part way through an expression evaluation. (A `setjmp` *macro* seems to help only if it expands to inline assembly code or a call to a special built-in function.) The temporaries may be correct on the initial call to `setjmp`, but are not likely to be on any return initiated by a corresponding call to `longjmp`. These considerations dictated the constraint that `setjmp` be called only from within fairly simple expressions, ones not likely to need temporary storage.

An alternative proposal considered by the Committee is to require that implementations recognize that calling `setjmp` is a special case,[4] and hence that they take whatever precautions are necessary to restore the `setjmp` environment properly upon a `longjmp` call. This proposal was rejected on grounds of consistency: implementations are currently *allowed* to implement library functions specially, but no other situations *require* special treatment.

4.6.2   Restore calling environment

4.6.2.1   The `longjmp` function

The Committee also considered requiring that a call to `longjmp` restore the (`setjmp`) calling environment fully — that upon execution of a `longjmp`, all local variables in the environment of `setjmp` have the values they did at the time of the `longjmp` call. Register variables create problems with this idea. Unfortunately, the best that many implementations attempt with register variables is to save them (in `jmp_buf`) at the time of the initial `setjmp` call, then restore them to that state on each return initiated by a `longjmp` call. Since compilers are certainly at liberty to change register variables to automatic, it is not obvious that a register declaration will indeed be rolled back. And since compilers are at liberty to change automatic variables to

---

[4]This proposal was considered prior to the adoption of the stricture that `setjmp` be a macro. It can be considered as equivalent to proposing that the `setjmp` macro expand to a call to a special built-in compiler function.

register (if their addresses are never taken), it is not obvious that an automatic declaration will *not* be rolled back. Hence the vague wording. In fact, the only reliable way to ensure that a local variable retain the value it had at the time of the call to `longjmp` is to define it with the `volatile` attribute.

Some implementations leave a process in a special state while a signal is being handled. An explicit reassurance must be given to the environment when the signal handler is done. To keep this job manageable, the Committee agreed to restrict `longjmp` to only one level of signal handling.

The `longjmp` function should not be called in an exit handler (i.e., a function registered with the `atexit` function (see §4.10.4.2)), since it might jump to some code which is no longer in scope.

4.7   Signal Handling
`<signal.h>`

This facility has been retained from the Base Document since the Committee felt it important to provide some standard mechanism for dealing with exceptional program conditions. Thus a subset of the signals defined in UNIX were retained in the Standard, along with the basic mechanisms of declaring signal handlers and (with adaptations, see §4.7.2.1) raising signals. For a discussion of the problems created by including signals, see §2.2.3.

The signal machinery contains many misnomers: `SIGFPE`, `SIGILL`, and `SIGSEGV` have their roots in PDP-11 hardware terminology, but the names are too entrenched to change. (The occurrence of `SIGFPE`, for instance, does not necessarily indicate a floating-point error.) A conforming implementation is not required to field *any* hardware interrupts.

The Committee has reserved the space of names beginning with `SIG` to permit implementations to add local names to `<signal.h>`. This implies that such names should not be otherwise used in a C source file which includes `<signal.h>`.

4.7.1   Specify signal handling

4.7.1.1   The `signal` function

When a signal occurs the normal flow of control of a program is interrupted. If a signal occurs that is being trapped by a signal handler, that handler is invoked. When it is finished, execution continues at the point at which the signal occurred. This arrangement could cause problems if the signal handler invokes a library function that was being executed at the time of the signal. Since library functions are not guaranteed to be re-entrant, they should not be called from a signal handler that returns. (See §2.2.3.) A specific exception to this rule has been granted for calls to `signal` from within the signal handler; otherwise, the handler could not reliably reset the signal.

The specification that some signals may be effectively set to `SIG_IGN` instead of `SIG_DFL` at program startup allows programs under UNIX systems to inherit this effective setting from parent processes.

For performance reasons, UNIX does not reset `SIGILL` to default handling when the handler is called (usually to emulate missing instructions). This treatment is sanctioned by specifying that whether reset occurs for `SIGILL` is *implementation-defined*.

4.7.2 Send signal

4.7.2.1 The `raise` function

The function `raise` replaces the Base Document's `kill` function. The latter has an extra argument which refers to the "process ID" affected by the signal. Since the execution model of the Standard does not deal with multi-processing, the Committee deemed it preferable to introduce a function which requires no (dummy) process argument. The Committee anticipates that IEEE 1003 will wish to standardize the `kill` function in the POSIX specification.

4.8 Variable Arguments `<stdarg.h>`

For a discussion of argument passing issues, see §3.7.1.

These macros, modeled after the UNIX `<varargs.h>` macros, have been added to enable the portable implementation in C of library functions such as `printf` and `scanf` (see §4.9.6). Such implementation could otherwise be difficult, considering newer machines that may pass arguments in machine registers rather than using the more traditional stack-oriented methods.

The definitions of these macros in the Standard differ from their forebears: they have been extended to support argument lists that have a fixed set of arguments preceding the variable list.

`va_start` and `va_arg` must exist as macros, since `va_start` uses an argument that is passed by name and `va_arg` uses an argument which is the name of a data type. Using `#undef` on these names leads to *undefined behavior*.

The `va_list` type is not necessarily assignable. However, a function can pass a pointer to its initialized argument list object, as noted below.

4.8.1 Variable argument list access macros

4.8.1.1 The `va_start` macro

`va_start` must be called within the body of the function whose argument list is to be traversed. That function can then pass a pointer to its `va_list` object `ap` to other functions to do the actual traversal. (It can, of course, traverse the list itself.)

The `parmN` argument to `va_start` is an aid to writing conforming ANSI C code for existing C implementations. Many implementations can use the second parameter within the structure of existing C language constructs to derive the address of the first variable argument. (Declaring `parmN` to be of storage class `register` would interfere with use of these constructs; hence the effect of such a declaration is *undefined behavior*. Other restrictions on the type of `parmN` are imposed for the same reason.) New implementations may choose to use hidden machinery that ignores the second argument to `va_start`, possibly even hiding a function call inside the macro.

Multiple `va_list` variables can be in use simulaneously in the same function; each requires its own calls to `va_start` and `va_end`.

4.8.1.2   The `va_arg` macro

Changing an arbitrary *type name* into a type name which is a pointer to that type could require sophisticated rewriting. To allow the implementation of `va_arg` as a macro, `va_arg` need only correctly handle those type names that can be transformed into the appropriate pointer type by appending a `*`, which handles most simple cases. (Typedefs can be defined to reduce more complicated types to a tractable form.) When using these macros it is important to remember that the type of an argument in a variable argument list will never be an integer type smaller than `int`, nor will it ever be `float`. (See §3.5.4.3.)

`va_arg` can only be used to access the value of an argument, not to obtain its address.

4.8.1.3   The `va_end` macro

`va_end` must also be called from within the body of the function having the variable argument list. In many implementations, this is a do-nothing operation; but those implementations that need it probably need it badly.

4.9   Input/Output
   `<stdio.h>`

Many implementations of the C runtime environment (most notably the UNIX operating system) provide, aside from the standard I/O library (`fopen`, `fclose`, `fread`, `fwrite`, `fseek`), a set of unbuffered I/O services (`open`, `close`, `read`, `write`, `lseek`). The Committee has decided not to standardize the latter set of functions.

A suggested semantics for these functions in the UNIX world may be found in the emerging IEEE P1003 standard. The standard I/O library functions use a *file pointer* for referring to the desired I/O stream. The unbuffered I/O services use a *file descriptor* (a small integer) to refer to the desired I/O stream.

Due to weak implementations of the standard I/O library, many implementors have assumed that the standard I/O library was used for small records and that the

unbuffered I/O library was used for large records. However, a good implementation of the standard I/O library can match the performance of the unbuffered services on large records. The user also has the capability of tuning the performance of the standard I/O library (with setvbuf) to suit the application.

Some subtle differences between the two sets of services can make the implementation of the unbuffered I/O services difficult:

- The model of a file used in the unbuffered I/O services is an array of characters. Many C environments do not support this file model.

- Difficulties arise when handling the new-line character. Many hosts use conventions other than an in-stream new-line character to mark the end of a line. The unbuffered I/O services assume that no translation occurs between the program's data and the file data when performing I/O, so either the new-line character translation would be lost (which breaks programs) or the implementor must be aware of the new-line translation (which results in non-portable programs).

- On UNIX systems, file descriptors 0, 1, and 2 correspond to the standard input, output, and error streams. This convention may be problematic for other systems in that (1) file descriptors 0, 1, and 2 may not be available or may be reserved for another purpose, (2) the operating system may use a different set of services for terminal I/O than file I/O.

In summary, the Committee chose not to standardize the unbuffered I/O services because:

- They duplicate the facilities provided by the standard I/O services.

- The performance of the standard I/O services can be the same or better than the unbuffered I/O services.

- The unbuffered I/O file model may not be appropriate for many C language environments.

4.9.1 Introduction

The macros _IOFBF, _IOLBF, _IONBF are enumerations of the third argument to setvbuf, a function adopted from UNIX System V.

SEEK_CUR, SEEK_END, and SEEK_SET have been moved to <stdio.h> from a header specified in the Base Document and not retained in the Standard.

FOPEN_MAX and TMP_MAX are added environmental limits of some interest to programs that manipulate multiple temporary files.

FILENAME_MAX is provided so that buffers to hold file names can be conveniently declared. If the target system supports arbitrarily long filenames, the implementor should provide some reasonable value (80?, 255?, 509?) rather than something unusable like USHRT_MAX.

4.9.2   Streams

C inherited its notion of text streams from the UNIX environment in which it was born. Having each line delimited by a single *new-line* character, regardless of the characteristics of the actual terminal, supported a simple model of text as a sort of arbitrary length scroll or "galley." Having a channel that is "transparent" (no file structure or reserved data encodings) eliminated the need for a distinction between text and binary streams.

Many other environments have different properties, however. If a program written in C is to produce a text file digestible by other programs, by text editors in particular, it must conform to the text formatting conventions of that environment.

The I/O facilities defined by the Standard are both more complex and more restrictive than the ancestral I/O facilities of UNIX. This is justified on pragmatic grounds: most of the differences, restrictions and omissions exist to permit C I/O implementations in environments which differ from the UNIX I/O model.

Troublesome aspects of the stream concept include:

The definition of lines. In the UNIX model, division of a file into lines is effected by new-line characters. Different techniques are used by other systems — lines may be separated by CR-LF (carriage return, line feed) or by unrecorded areas on the recording medium, or each line may be prefixed by its length. The Standard addresses this diversity by specifying that new-line be used as a line separator at the program level, but then permitting an implementation to transform the data read or written to conform to the conventions of the environment.

Some environments represent text lines as blank-filled fixed-length records. Thus the Standard specifies that it is *implementation-defined* whether trailing blanks are removed from a line on input. (This specification also addresses the problems of environments which represent text as variable-length records, but do not allow a record length of 0: an empty line may be written as a one-character record containing a blank, and the blank is stripped on input.)

Transparency. Some programs require access to external data without modification. For instance, transformation of CR-LF to new-line character is usually not desirable when object code is processed. The Standard defines two stream types, *text* and *binary*, to allow a program to define, when a file is opened, whether the preservation of its exact contents or of its line structure is more important in an environment which cannot accurately reflect both.

Random access. The UNIX I/O model features random access to data in a file, indexed by character number. On systems where a new-line character processed by the program represents an unknown number of physically recorded characters, this simple mechanism cannot be consistently supported for text streams. The Standard abstracts the significant properties of random access for text streams: the ability to determine the current file position and then

later reposition the file to the same location. `ftell` returns a *file position indicator*, which has no necessary interpretation except that an `fseek` operation with that indicator value will position the file to the same place. Thus an implementation may encode whatever file positioning information is most appropriate for a text file, subject only to the constraint that the encoding be representable as a `long`. Use of `fgetpos` and `fsetpos` removes even this constraint.

Buffering. UNIX allows the program to control the extent and type of buffering for various purposes. For example, a program can provide its own large I/O buffer to improve efficiency, or can request unbuffered terminal I/O to process each input character as it is entered. Other systems do not necessarily support this generality. Some systems provide only line-at-a-time access to terminal input; some systems support program-allocated buffers only by copying data to and from system-allocated buffers for processing. Buffering is addressed in the Standard by specifying UNIX-like `setbuf` and `setvbuf` functions, but permitting great latitude in their implementation. A conforming library need neither attempt the impossible nor respond to a program attempt to improve efficiency by introducing additional overhead.

Thus, the Standard imposes a clear distinction between *text streams*, which must be mapped to suit local custom, and *binary streams*, for which no mapping takes place. Local custom on UNIX (and related) systems is of course to treat the two sorts of streams identically, and nothing in the Standard requires any changes to this practice.

Even the specification of binary streams requires some changes to accommodate a wide range of systems. Because many systems do not keep track of the length of a file to the nearest byte, an arbitrary number of characters may appear on the end of a binary stream directed to a file. The Standard cannot forbid this implementation, but does require that this padding consist only of null characters. The alternative would be to restrict C to producing binary files digestible only by other C programs; this alternative runs counter to the *spirit of C*.

The set of characters required to be preserved in text stream I/O are those needed for writing C programs; the intent is the Standard should permit a C translator to be written in a maximally portable fashion. Control characters such as backspace are not required for this purpose, so their handling in text streams is not mandated.

It was agreed that some minimum maximum line length must be mandated; 254 was chosen.

4.9.3   Files

The *as if* principle is once again invoked to define the nature of input and output in terms of just two functions, `fgetc` and `fputc`. The actual primitives in a given system may be quite different.

Buffering, and unbuffering, is defined in a way suggesting the desired interactive behavior; but an implementation may still be conforming even if delays (in a network or terminal controller) prevent output from appearing in time. It is the *intent* that matters here.

No constraints are imposed upon file names, except that they must be representable as strings (with no embedded null characters).

4.9.4   Operations on files

4.9.4.1   The `remove` function

The Base Document provides the `unlink` system call to remove files. The UNIX-specific definition of this function prompted the Committee to replace it with a portable function.

4.9.4.2   The `rename` function

This function has been added to provide a system-independent atomic operation to change the name of an existing file; the Base Document only provided the `link` system call, which gives the file a new name without removing the old one, and which is extremely system-dependent.

The Committee considered a proposal that `rename` should quietly copy a file if simple renaming couldn't be performed in some context, but rejected this as potentially too expensive at execution time.

`rename` is meant to give access to an underlying facility of the execution environment's operating system. When the new name is the name of an existing file, some systems allow the renaming (and delete the old file or make it inaccessible by that name), while others prohibit the operation. The effect of `rename` is thus *implementation-defined*.

4.9.4.3   The `tmpfile` function

The `tmpfile` function is intended to allow users to create binary "scratch" files. The *as if* principle implies that the information in such a file need never actually be stored on a file-structured device.

The temporary file is created in binary update mode, because it will presumably be first written and then read as transparently as possible. Trailing null-character padding may cause problems for some existing programs.

4.9.4.4   The `tmpnam` function

This function allows for more control than `tmpfile`: a file can be opened in binary mode or text mode, and files are not erased at completion.

There is always some time between the call to `tmpnam` and the use (in `fopen`) of the returned name. Hence it is conceivable that in some implementations the name, which named no file at the call to `tmpnam`, has been used as a filename by the time of

the call to `fopen`. Implementations should devise name-generation strategies which minimize this possibility, but users should allow for this possibility.

4.9.5 File access functions

4.9.5.1 The `fclose` function

On some operating systems it is difficult, or impossible, to create a file unless something is written to the file. A maximally portable program which relies on a file being created must write something to the associated stream before closing it.

4.9.5.2 The `fflush` function

The `fflush` function ensures that output has been forced out of internal I/O buffers for a specified stream. Occasionally, however, it is necessary to ensure that *all* output is forced out, and the programmer may not conveniently be able to specify all the currently-open streams (perhaps because some streams are manipulated within library packages).[5] To provide an implementation-independent method of flushing all output buffers, the Standard specifies that this is the result of calling `fflush` with a NULL argument.

4.9.5.3 The `fopen` function

The `b` type modifier has been added to deal with the text/binary dichotomy (see §4.9.2). Because of the limited ability to seek within text files (see §4.9.9.1), an implementation is at liberty to treat the old update + modes as if `b` were also specified. Table 4.1 tabulates the capabilities and actions associated with the various specified mode string arguments to `fopen`.

Table 4.1: File and stream properties of `fopen` modes

	r	w	a	r+	w+	a+
file must exist before open	√			√		
old file contents discarded on open		√			√	
stream can be read	√			√	√	√
stream can be written		√	√	√	√	√
stream can be written only at end			√			√

Other specifications for files, such as record length and block size, are not specified in the Standard, due to their widely varying characteristics in different operating

---

[5] For instance, on a system (such as UNIX) which supports process forks, it is usually necessary to flush all output buffers just prior to the fork.

environments. Changes to file access modes and buffer sizes may be specified using the `setvbuf` function. (See §4.9.5.6.) An implementation may choose to allow additional file specifications as part of the `mode` string argument. For instance,

```
file1 = fopen(file1name,"wb,reclen=80");
```

might be a reasonable way, on a system which provides record-oriented binary files, for an implementation to allow a programmer to specify record length.

A change of input/output direction on an update file is only allowed following a `fsetpos`, `fseek`, `rewind`, or `fflush` operation, since these are precisely the functions which assure that the I/O buffer has been flushed.

The Standard (§4.9.2) imposes the requirement that binary files not be truncated when they are updated. This rule does not preclude an implementation from supporting additional file types that do truncate when written to, even when they are opened with the same sort of `fopen` call. Magnetic tape files are an example of a file type that must be handled this way. (On most tape hardware it is impossible to write to a tape without destroying immediately following data.) Hence tape files are not "binary files" within the meaning of the Standard. A conforming hosted implementation must provide (and document) at least one file type (on disk, most likely) that behaves exactly as specified in the Standard.

4.9.5.4   The `freopen` function

4.9.5.5   The `setbuf` function

`setbuf` is subsumed by `setvbuf`, but has been retained for compatibility with old code.

4.9.5.6   The `setvbuf` function

`setvbuf` has been adopted from UNIX System V, both to control the nature of stream buffering and to specify the size of I/O buffers. An implementation is not required to make actual use of a buffer provided for a stream, so a program must never expect the buffer's contents to reflect I/O operations. Further, the Standard does not require that the requested buffering be implemented; it merely mandates a standard mechanism for requesting whatever buffering services might be provided.

Although three types of buffering are defined, an implementation may choose to make one or more of them equivalent. For example, a library may choose to implement line-buffering for binary files as equivalent to unbuffered I/O or may choose to always implement full-buffering as equivalent to line-buffering.

The general principle is to provide portable code with a means of requesting the most appropriate popular buffering style, but not to require an implementation to support these styles.

4.9.6   Formatted input/output functions

4.9.6.1   The fprintf function

Use of the L modifier with floating conversions has been added to deal with formatted output of the new type long double.

Note that the %X and %x formats expect a corresponding int argument; %lX or %lx must be supplied with a long int argument.

The conversion specification %p has been added for pointer conversion, since the size of a pointer is not necessarily the same as the size of an int. Because an implementation may support more than one size of pointer, the corresponding argument is expected to be a (void *) pointer.

The %n format has been added to permit ascertaining the number of characters converted up to that point in the current invocation of the formatter.

Some pre-Standard implementations switch formats for %g at an exponent of −3 instead of (the Standard's) −4: existing code which requires the format switch at −3 will have to be changed.

Some existing implementations provide %D and %O as synonyms or replacements for %ld and %lo. The Committee considered the latter notation preferable.

The Committee has reserved lower case conversion specifiers for future standardization.

The use of leading zero in field widths to specify zero padding has been superseded by a precision field. The older mechanism has been retained.

Some implementations have provided the format %r as a means of indirectly passing a variable-length argument list. The functions vfprintf, etc., are considered to be a more controlled method of effecting this indirection, so %r was not adopted in the Standard. (See §4.9.6.7.)

The printing formats for numbers is not entirely specified. The requirements of the Standard are loose enough to allow implementations to handle such cases as signed zero, *not-a-number*, and *infinity* in an appropriate fashion.

4.9.6.2   The fscanf function

The specification of fscanf is based in part on these principles:

- As soon as one specified conversion fails, the whole function invocation fails.

- One-character pushback is sufficient for the implementation of fscanf. Given the invalid field "-.x", the characters "-." are not pushed back.

- If a "flawed field" is detected, no value is stored for the corresponding argument.

- The conversions performed by fscanf are compatible with those performed by strtod and strtol.

Input pointer conversion with %p has been added, although it is obviously risky, for symmetry with fprintf. The %i format has been added to permit the scanner to determine the radix of the number in the input stream; the %n format has been added to make available the number of characters scanned thus far in the current invocation of the scanner.

White space is now defined by the isspace function. (See §4.3.1.9.)

An implementation must not use the ungetc function to perform the necessary one-character pushback. In particular, since the unmatched text is left "unread," the file position indicator as reported by the ftell function must be the position of the character remaining to be read. Furthermore, if the unread characters were themselves pushed back via ungetc calls, the pushback in fscanf must not affect the push-back stack in ungetc. A scanf call that matches N characters from a stream must leave the stream in the same state as if N consecutive getc calls had been issued.

4.9.6.3   The printf function

See comments of section §4.9.6.1 above.

4.9.6.4   The scanf function

See comments in section §4.9.6.2 above.

4.9.6.5   The sprintf function

See §4.9.6.1 for comments on output formatting.

In the interests of minimizing redundancy, sprintf has subsumed the older, rather uncommon, ecvt, fcvt, and gcvt.

4.9.6.6   The sscanf function

The behavior of sscanf on encountering end of string has been clarified. See also comments in section §4.9.6.2 above.

4.9.6.7   The vfprintf function

The functions vfprintf, vprintf, and vsprintf have been adopted from UNIX System V to facilitate writing special purpose formatted output functions.

4.9.6.8   The vprintf function

See §4.9.6.7.

4.9.6.9   The vsprintf function

See §4.9.6.7.

4.9.7 Character input/output functions

4.9.7.1 The `fgetc` function

Because much existing code assumes that `fgetc` and `fputc` are the actual functions equivalent to the macros `getc` and `putc`, the Standard requires that they not be implemented as macros.

4.9.7.2 The `fgets` function

This function subsumes `gets`, which has no limit to prevent storage overwrite on arbitrary input (see §4.9.7.7).

4.9.7.3 The `fputc` function

See §4.9.7.1.

4.9.7.4 The `fputs` function

4.9.7.5 The `getc` function

`getc` and `putc` have often been implemented as unsafe macros, since it is difficult in such a macro to touch the `stream` argument only once. Since this danger is common in prior art, these two functions are explicitly permitted to evaluate `stream` more than once.

4.9.7.6 The `getchar` function

4.9.7.7 The `gets` function

See §4.9.7.2.

4.9.7.8 The `putc` function

See §4.9.7.5.

4.9.7.9 The `putchar` function

4.9.7.10 The `puts` function

`puts(s)` is not exactly equivalent to `fputs(stdout,s)`; `puts` also writes a new line after the argument string. This incompatibility reflects existing practice.

4.9.7.11 The `ungetc` function

The Base Document requires that at least one character be read before `ungetc` is called, in certain implementation-specific cases. The Committee has removed this requirement, thus obliging a `FILE` structure to have room to store one character of

pushback regardless of the state of the buffer; it felt that this degree of generality makes clearer the ways in which the function may be used.

It is permissible to push back a different character than that which was read; this accords with common existing practice. The last-in, first-out nature of `ungetc` has been clarified.

`ungetc` is typically used to handle algorithms, such as tokenization, which involve one-character lookahead in text files. `fseek` and `ftell` are used for random access, typically in binary files. So that these disparate file-handling disciplines are not unnecessarily linked, the value of a text file's file position indicator immediately after `ungetc` has been specified as indeterminate.

Existing practice relies on two different models of the effect of `ungetc`. One model can be characterized as writing the pushed-back character "on top of" the previous character. This model implies an implementation in which the pushed-back characters are stored within the file buffer and bookkeeping is performed by setting the file position indicator to the previous character position. (Care must be taken in this model to recover the overwritten character values when the pushed-back characters are discarded as a result of other operations on the stream.) The other model can be characterized as pushing the character "between" the current character and the previous character. This implies an implementation in which the pushed-back characters are specially buffered (within the FILE structure, say) and accounted for by a flag or count. In this model it is natural *not* to move the file position indicator. The indeterminacy of the file position indicator while pushed-back characters exist accommodates both models.

Mandating either model (by specifying the effect of `ungetc` on a text file's file position indicator) creates problems with implementations that have assumed the other model. Requiring the file position indicator not to change after `ungetc` would necessitate changes in programs which combine random access and tokenization on text files, and rely on the file position indicator marking the end of a token even after pushback. Requiring the file position indicator to back up would create severe implementation problems in certain environments, since in some file organizations it can be impossible to find the previous input character position without having read the file sequentially to the point in question.[6]

4.9.8   Direct input/output functions

4.9.8.1   The `fread` function

`size_t` is the appropriate type both for an object size and for an array bound (see

---

[6]Consider, for instance, a sequential file of variable-length records in which a line is represented as a count field followed by the characters in the line. The file position indicator must encode a character position as the position of the count field plus an offset into the line; from the position of the count field and the length of the line, the next count field can be found. Insufficient information is available for finding the *previous* count field, so backing up from the first character of a line necessitates, in the general case, a sequential read from the start of the file.

§3.3.3.4), so this is the type of `size` and `nelem`.

4.9.8.2   The `fwrite` function

See §4.9.8.1.

4.9.9   File positioning functions

4.9.9.1   The `fgetpos` function

`fgetpos` and `fsetpos` have been added to allow random access operations on files which are too large to handle with `fseek` and `ftell`.

4.9.9.2   The `fseek` function

Whereas a binary file can be treated as an ordered sequence of bytes, counting from zero, a text file need not map one-to-one to its internal representation (see §4.9.2). Thus, only seeks to an earlier reported position are permitted for text files. The need to encode both record position and position within a record in a `long` value may constrain the size of text files upon which `fseek-ftell` can be used to be considerably smaller than the size of binary files.

Given these restrictions, the Committee still felt that this function has enough utility, and is used in sufficient existing code, to warrant its retention in the Standard. `fgetpos` and `fsetpos` have been added to deal with files which are too large to handle with `fseek` and `ftell`.

The `fseek` function will reset the end-of-file flag for the stream; the error flag is not changed unless an error occurs, when it will be set.

4.9.9.3   The `fsetpos` function

4.9.9.4   The `ftell` function

`ftell` can fail for at least two reasons:

- the stream is associated with a terminal, or some other file type for which *file position indicator* is meaningless; or

- the file may be positioned at a location not representable in a `long int`.

Thus a method for `ftell` to report failure has been specified.
See also §4.9.9.1.

4.9.9.5   The `rewind` function

Resetting the end-of-file and error indicators was added to the specification of `rewind` to make the specification more logically consistent.

4.9.10   Error-handling functions

4.9.10.1   The `clearerr` function

4.9.10.2   The `feof` function

4.9.10.3   The `ferror` function

4.9.10.4   The `perror` function

At various times, the Committee considered providing a form of `perror` that delivers up an error string version of `errno` without performing any output. It ultimately decided to provide this capability in a separate function, `strerror`. (See §4.11.6.1).

4.10   General Utilities
`<stdlib.h>`

The header `<stdlib.h>` was invented by the Committee to hold an assortment of functions that were otherwise homeless.

4.10.1   String conversion functions

4.10.1.1   The `atof` function

`atof`, `atoi`, and `atol` are subsumed by `strtod` and `strtol`, but have been retained because they are used extensively in existing code. They are less reliable, but may be faster if the argument is known to be in a valid range.

4.10.1.2   The `atoi` function

See §4.10.1.1.

4.10.1.3   The `atol` function

See §4.10.1.1.

4.10.1.4   The `strtod` function

`strtod` and `strtol` have been adopted (from UNIX System V) because they offer more control over the conversion process, and because they are required not to produce unexpected results on overflow during conversion.

4.10.1.5   The `strtol` function

See §4.10.1.4.

4.10.1.6 The strtoul function

strtoul was introduced by the Committee to provide a facility like strtol for
unsigned long values. Simply using strtol in such cases could result in overflow
upon conversion.

4.10.2 Pseudo-random sequence generation functions

4.10.2.1 The rand function

The Committee decided that an implementation should be allowed to provide a rand
function which generates the best random sequence possible in that implementation,
and therefore mandated no standard algorithm. It recognized the value, however,
of being able to generate the same pseudo-random sequence in different implemen-
tations, and so it has published as an example in the Standard an algorithm that
generates the same pseudo-random sequence in any conforming implementation,
given the same seed.

4.10.2.2 The srand function

4.10.3 Memory management functions

The treatment of null pointers and 0-length allocation requests in the definition of
these functions was in part guided by a desire to support this paradigm:

```
OBJ * p; /* pointer to a variable list of OBJ's */

/* initial allocation */
p = (OBJ *) calloc(0, sizeof(OBJ));
/* ... */

/* reallocations until size settles */
while(/* list changes size to c */) {
    p = (OBJ *) realloc((void *)p, c*sizeof(OBJ));
    /* ... */
}
```

This coding style, not necessarily endorsed by the Committee, is reported to be in
widespread use.

Some implementations have returned non-null values for allocation requests of
0 bytes. Although this strategy has the theoretical advantage of distinguishing be-
tween "nothing" and "zero" (an unallocated pointer vs. a pointer to zero-length
space), it has the more compelling theoretical disadvantage of requiring the concept
of a zero-length object. Since such objects cannot be declared, the only way they
could come into existence would be through such allocation requests. The Com-
mittee has decided not to accept the idea of zero-length objects. The allocation

functions may therefore return a null pointer for an allocation request of zero bytes. Note that this treatment does not preclude the paradigm outlined above.

QUIET CHANGE

A program which relies on size-0 allocation requests returning a non-null pointer will behave differently.

Some implementations provide a function (often called `alloca`) which allocates the requested object from automatic storage; the object is automatically freed when the calling function exits. Such a function is not efficiently implementable in a variety of environments, so it was not adopted in the Standard.

4.10.3.1   The `calloc` function

Both `nelem` and `elsize` must be of type `size_t`, for reasons similar to those for `fread` (see §4.9.8.1).

If a scalar with all bits zero is not interpreted as a zero value by an implementation, then `calloc` may have astonishing results in existing programs transported there.

4.10.3.2   The `free` function

The Standard makes clear that a program may only free that which has been allocated, that an allocation may only be freed once, and that a region may not be accessed once it is freed. Some implementations allow more dangerous license. The null pointer is specified as a valid argument to this function to reduce the need for special-case coding.

4.10.3.3   The `malloc` function

4.10.3.4   The `realloc` function

A null first argument is permissible. If the first argument is not null, and the second argument is 0, then the call frees the memory pointed to by the first argument, and a null argument may be returned; this specification is consistent with the policy of not allowing zero-size objects.

4.10.4   Communication with the environment

4.10.4.1   The `abort` function

The Committee vacillated over whether a call to `abort` should return if the signal `SIGABRT` is caught or ignored. To minimize astonishment, the final decision was that `abort` never returns.

4.10.4.2 The atexit function

`atexit` provides a program with a convenient way to clean up the environment before it exits. It is adapted from the Whitesmiths C run-time library function `onexit`.

A suggested alternative was to use the `SIGTERM` facility of the signal/raise machinery, but that would not give the last-in first-out stacking of multiple functions so useful with `atexit`.

It is the responsibility of the library to maintain the chain of registered functions so that they are invoked in the correct sequence upon program exit.

4.10.4.3 The exit function

The argument to `exit` is a status indication returned to the invoking environment. In the UNIX operating system, a value of 0 is the successful return code from a program. As usage of C has spread beyond UNIX, `exit(0)` has often been retained as an idiom indicating successful termination, even on operating systems with different systems of return codes. This usage is thus recognized as standard. There has never been a portable way of indicating a non-successful termination, since the arguments to `exit` are then implementation-defined. The macro `EXIT_FAILURE` has been added to provide such a capability. (`EXIT_SUCCESS` has been added as well.)

Aside from calls explicitly coded by a programmer, `exit` is invoked on return from `main`. Thus in at least this case, the body of `exit` cannot assume the existence of any objects with automatic storage duration (except those declared in `exit`).

4.10.4.4 The getenv function

The definition of `getenv` is designed to accommodate both implementations that have all in-memory read-only environment strings and those that may have to read an environment string into a static buffer. Hence the pointer returned by the `getenv` function points to a string not modifiable by the caller. If an attempt is made to change this string, the behavior of future calls to `getenv` is undefined.

A corresponding `putenv` function was omitted from the Standard, since its utility outside a multi-process environment is questionable, and since its definition is properly the domain of an operating system standard.

4.10.4.5 The system function

The `system` function allows a program to suspend its execution temporarily in order to run another program to completion.

Information may be passed to the called program in three ways: through command-line argument strings, through the environment, and (most portably) through data files. Before calling the system function, the calling program should close all such data files.

Information may be returned from the called program in two ways: through the implementation-defined return value (in many implementations, the termination status code which is the argument to the `exit` function is returned by the implementation to the caller as the value returned by the `system` function), and (most portably) through data files.

If the environment is interactive, information may also be exchanged with users of interactive devices.

Some implementations offer built-in programs called "commands" (for example, "date") which may provide useful information to an application program via the `system` function. The Standard does not attempt to characterize such commands, and their use is not portable.

On the other hand, the use of the `system` function *is* portable, provided the implementation supports the capability. The Standard permits the application to ascertain this by calling the `system` function with a null pointer argument. Whether more levels of nesting are supported can also be ascertained this way; assuming more than one such level is obviously dangerous.

4.10.5   Searching and sorting utilities

4.10.5.1   The `bsearch` function

4.10.5.2   The `qsort` function

4.10.6   Integer arithmetic functions

`abs` was moved from `<math.h>` as it was the only function in that library which did not involve `double` arithmetic. Some programs have included `<math.h>` solely to gain access to `abs`, but in some implementations this results in unused floating-point run-time routines becoming part of the translated program.

4.10.6.1   The `abs` function

The Committee rejected proposals to add an absolute value operator to the language. An implementation can provide a built-in function for efficiency.

4.10.6.2   The `div` function

`div` and `ldiv` provide a well-specified semantics for signed integral division and remainder operations. The semantics were adopted to be the same as in FORTRAN. Since these functions return both the quotient and the remainder, they also serve as a convenient way of efficiently modelling underlying hardware that computes both results as part of the same operation. Table 4.2 summarizes the semantics of these functions.

Divide-by-zero is described as *undefined behavior* rather than as setting **errno** to `EDOM`. The program can as easily check for a zero divisor before a division as for an error code afterwards, and the adopted scheme reduces the burden on the function.

Table 4.2: Results of `div` and `ldiv`

numer	denom	quot	rem
7	3	2	1
−7	3	−2	−1
7	−3	−2	1
−7	−3	2	−1

4.10.6.3   The `labs` function

4.10.6.4   The `ldiv` function

4.10.7   Multibyte character functions

See §2.2.1.2 for an overall discussion of multibyte character representations and wide characters.

4.10.7.1   The `mblen` function

4.10.7.2   The `mbtowc` function

4.10.7.3   The `wctomb` function

4.10.8   Multibyte string functions

See §2.2.1.2 for an overall discussion of multibyte character representations and wide characters.

4.10.8.1   The `mbstowcs` function

4.10.8.2   The `wcstombs` function

4.11   STRING HANDLING
   <string.h>

The Committee felt that the functions in this section were all excellent candidates for replacement by high-performance built-in operations. Hence many simple functions have been retained, and several added, just to leave the door open for better implementations of these common operations.

   The Standard reserves function names beginning with `str` or `mem` for possible future use.

4.11.1   String function conventions

`memcpy`, `memset`, `memcmp`, and `memchr` have been adopted from several existing implementations. The general goal was to provide equivalent capabilities for three

types of byte sequences:

- null-terminated strings (`str-`),

- null-terminated strings with a maximum length (`strn-`), and

- transparent data of specified length (`mem-`).

4.11.2   Copying functions

A block copy routine should be "right": it should work correctly even if the blocks being copied overlap. Otherwise it is more difficult to correctly code such overlapping copy operations, and portability suffers because the optimal C-coded algorithm on one machine may be horribly slow on another.

A block copy routine should be "fast": it should be implementable as a few inline instructions which take maximum advantage of any block copy provisions of the hardware. Checking for overlapping copies produces too much code for convenient inlining in many implementations. The programmer knows in a great many cases that the two blocks cannot possibly overlap, so the space and time overhead are for naught.

These arguments are contradictory but each is compelling. Therefore the Standard mandates two block copy functions: `memmove` is required to work correctly even if the source and destination overlap, while `memcpy` can presume nonoverlapping operands and be optimized accordingly.

4.11.2.1   The `memcpy` function

4.11.2.2   The `memmove` function

4.11.2.3   The `strcpy` function

4.11.2.4   The `strncpy` function

`strncpy` was initially introduced into the C library to deal with fixed-length name fields in structures such as directory entries. Such fields are not used in the same way as strings: the trailing null is unnecessary for a maximum-length field, and setting trailing bytes for shorter names to null assures efficient field-wise comparisons. `strncpy` is not by origin a "bounded `strcpy`," and the Committee has preferred to recognize existing practice rather than alter the function to better suit it to such use.

4.11.3   Concatenation functions

4.11.3.1   The `strcat` function

4.11.3.2   The `strncat` function

Note that this function may add n+1 characters to the string.

4.11.4   Comparison functions

4.11.4.1   The memcmp function

See §4.11.1.

4.11.4.2   The strcmp function

4.11.4.3   The strcoll function

strcoll and strxfrm provide for *locale-specific* string sorting. strcoll is intended for applications in which the number of comparisons is small; strxfrm is more appropriate when items are to be compared a number of times — the cost of transformation is then only paid once.

4.11.4.4   The strncmp function

4.11.4.5   The strxfrm function

See §4.11.4.3.

4.11.5   Search functions

4.11.5.1   The memchr function

See §4.11.1.

4.11.5.2   The strchr function

4.11.5.3   The strcspn function

4.11.5.4   The strpbrk function

4.11.5.5   The strrchr function

4.11.5.6   The strspn function

4.11.5.7   The strstr function

The strstr function is an invention of the Committee. It is included as a hook for efficient substring algorithms, or for built-in substring instructions.

4.11.5.8   The strtok function

This function has been included to provide a convenient solution to many simple problems of lexical analysis, such as scanning command line arguments.

RATIONALE

4.11.6   Miscellaneous functions

4.11.6.1   The `memset` function

See §4.11.1, and §4.10.3.1.

4.11.6.2   The `strerror` function

This function is a descendant of `perror` (see §4.9.10.4). It is defined such that it can return a pointer to an in-memory read-only string, or can copy a string into a static buffer on each call.

4.11.6.3   The `strlen` function

This function is now specified as returning a value of type `size_t`. (See §3.3.3.4.)

4.12   DATE AND TIME
`<time.h>`

4.12.1   Components of time

The types `clock_t` and `time_t` are arithmetic because values of these types must, in accordance with existing practice, on occasion be compared with -1 (a "don't-know" indication) suitably cast. No arithmetic properties of these types are defined by the Standard, however, in order to allow implementations the maximum flexibility in choosing ranges, precisions, and representations most appropriate to their intended application. The representation need not be a count of some basic unit; an implementation might conceivably represent different components of a temporal value as subfields of an integral type.

Many C environments do not support the Base Document library concepts of daylight savings or time zones. Both notions are defined geographically and politically, and thus may require more knowledge about the real world than an implementation can support. Hence the Standard specifies the date and time functions such that information about DST and time zones is not required. The Base Document function `tzset`, which would require dealing with time zones, has been excluded altogether. An implementation reports that information about DST is not available by setting the `tm_isdst` field in a broken-down time to a negative value. An implementation may return a null pointer from a call to `gmtime` if information about the displacement between Universal Time (*née* GMT) and local time is not available.

4.12.2   Time manipulation functions

4.12.2.1   The `clock` function

The function is intended for measuring intervals of execution time, in whatever units an implementation desires. The conflicting goals of high resolution, long interval

capacity, and low timer overhead must be balanced carefully in the light of this intended use.

4.12.2.2   The difftime function

difftime is an invention of the Committee. It is provided so that an implementation can store an indication of the date/time value in the most efficient format possible and still provide a method of calculating the difference between two times.

4.12.2.3   The mktime function

mktime was invented by the Committee to complete the set of time functions. With this function it becomes possible to perform portable calculations involving clock times and broken-down times.

The rules on the ranges of the fields within the *timeptr record are crafted to permit useful arithmetic to be done. For instance, here is a paradigm for continuing some loop for an hour:

```
#include <time.h>
struct tm    when;
time_t       now;
time_t       deadline;

/* ... */
now = time(0);
when = *localtime(&now);
when.tm_hour += 1;   /* result is in the range [1,24] */
deadline = mktime(&when);

printf("Loop will finish: %s\n", asctime(&when));
while ( difftime(deadline,time(0)) > 0 ) whatever();
```

The specification of mktime guarantees that the addition to the tm_hour field produces the correct result even when the new value of tm_hour is 24, i.e., a value outside the range ever returned by a library function in a struct tm object.

One of the reasons for adding this function is to replace the capability to do such arithmetic which is lost when a programmer cannot depend on time_t being an integral multiple of some known time unit.

Several readers of earlier versions of this Rationale have pointed out apparent problems in this example if now is just before a transition into or out of daylight savings time. However, when.tm_isdst indicates what sort of time was the basis of the calculation. Implementors, take heed. If this field is set to −1 on input, one truly ambiguous case involves the transition out of daylight savings time. As DST is currently legislated in the USA, the hour 0100–0159 occurs twice, first as DST and then as standard time. Hence an unlabeled 0130 on this date is problematic.

An implementation may choose to take this as DST or standard time, marking its decision in the `tm_isdst` field. It may also legitimately take this as invalid input (and return `(time_t)(-1)`).

4.12.2.4   The `time` function

Since no measure is given for how precise an implementation's *best approximation* to the current time must be, an implementation could always return the same date, instead of a more honest −1. This is, of course, not the intent.

4.12.3   Time conversion functions

4.12.3.1   The `asctime` function

Although the name of this function suggests a conflict with the principle of removing ASCII dependencies from the Standard, the name has been retained due to prior art. For the same reason of existing practice, a proposal to remove the newline character from the string format was not adopted. Proposals to allow for the use of languages other than English in naming weekdays and months met with objections on grounds of prior art, and on grounds that a truly international version of this function was difficult to specify: three-letter abbreviation of weekday and month names is not universally conventional, for instance. The `strftime` function (§4.12.3.5) provides appropriate facilities for locale-specific date and time strings.

4.12.3.2   The `ctime` function

4.12.3.3   The `gmtime` function

This function has been retained, despite objections that GMT — that is, Coordinated Universal Time (UTC) — is not available in some implementations, since UTC is a useful and widespread standard representation of time. If UTC is not available, a null pointer may be returned.

4.12.3.4   The `localtime` function

4.12.3.5   The `strftime` function

`strftime` provides a way of formatting the date and time in the appropriate locale-specific fashion, using the `%c`, `%x`, and `%X` format specifiers. More generally, it allows the programmer to tailor whatever date and time format is appropriate for a given application. The facility is based on the UNIX system `date` command. See §4.4 for further discussion of locale specification.

For the field controlled by `%P`, an implementation may wish to provide special symbols to mark noon and midnight.

4.13   Future library directions

4.13.1   Errors `<errno.h>`

4.13.2   Character handling `<ctype.h>`

4.13.3   Localization `<locale.h>`

4.13.4   Mathematics `<math.h>`

4.13.5   Signal handling `<signal.h>`

4.13.6   Input/output `<stdio.h>`

4.13.7   General utilities `<stdlib.h>`

4.13.8   String handling `<string.h>`

Section 5

APPENDICES

Most of the material in the appendices is not new. It is simply a summary of information in the Standard, collated for the convenience of users of the Standard.

New (advisory) information is found in Appendix E (Common Warnings) and in Appendix F.5 (Common Extensions). The section on common extensions is provided in part to give programmers even further information which may be useful in avoiding features of local dialects of C.

Index