

BS ISO/IEC 9899:2011
Incorporating corrigendum July 2012



BSI Standards Publication

Information technology — Programming languages — C

bsi.

...making excellence a habit.
AVA Documents™
Document Sharing Hub

National foreword

This British Standard is the UK implementation of ISO/IEC 9899:2011, incorporating corrigendum July 2012. It supersedes BS ISO/IEC 9899:2011, which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environments and system software interfaces.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2013.

Published by BSI Standards Limited 2013

ISBN 978 0 580 80165 5

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 March 2013.

Amendments/corrigenda issued since publication

Date	Text affected
31 March 2013	Implementation of ISO/IEC corrigendum July 2012: Macro names in Subclauses 6.10.8.1 and 6.10.8.3 modified

Information technology — Programming languages — C

Technologies de l'information — Langages de programmation — C





COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2011

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Foreword	xiii
Introduction	xvii
1. Scope	1
2. Normative references	2
3. Terms, definitions, and symbols	3
4. Conformance	8
5. Environment	10
5.1 Conceptual models	10
5.1.1 Translation environment	10
5.1.2 Execution environments	12
5.2 Environmental considerations	22
5.2.1 Character sets	22
5.2.2 Character display semantics	24
5.2.3 Signals and interrupts	25
5.2.4 Environmental limits	25
6. Language	35
6.1 Notation	35
6.2 Concepts	35
6.2.1 Scopes of identifiers	35
6.2.2 Linkages of identifiers	36
6.2.3 Name spaces of identifiers	37
6.2.4 Storage durations of objects	38
6.2.5 Types	39
6.2.6 Representations of types	44
6.2.7 Compatible type and composite type	47
6.2.8 Alignment of objects	48
6.3 Conversions	50
6.3.1 Arithmetic operands	50
6.3.2 Other operands	54
6.4 Lexical elements	57
6.4.1 Keywords	58
6.4.2 Identifiers	59
6.4.3 Universal character names	61
6.4.4 Constants	62
6.4.5 String literals	70
6.4.6 Punctuators	72
6.4.7 Header names	73
6.4.8 Preprocessing numbers	74
6.4.9 Comments	75

6.5	Expressions	76
6.5.1	Primary expressions	78
6.5.2	Postfix operators	79
6.5.3	Unary operators	88
6.5.4	Cast operators	91
6.5.5	Multiplicative operators	92
6.5.6	Additive operators	92
6.5.7	Bitwise shift operators	94
6.5.8	Relational operators	95
6.5.9	Equality operators	96
6.5.10	Bitwise AND operator	97
6.5.11	Bitwise exclusive OR operator	98
6.5.12	Bitwise inclusive OR operator	98
6.5.13	Logical AND operator	99
6.5.14	Logical OR operator	99
6.5.15	Conditional operator	100
6.5.16	Assignment operators	101
6.5.17	Comma operator	105
6.6	Constant expressions	106
6.7	Declarations	108
6.7.1	Storage-class specifiers	109
6.7.2	Type specifiers	111
6.7.3	Type qualifiers	121
6.7.4	Function specifiers	125
6.7.5	Alignment specifier	127
6.7.6	Declarators	128
6.7.7	Type names	136
6.7.8	Type definitions	137
6.7.9	Initialization	139
6.7.10	Static assertions	145
6.8	Statements and blocks	146
6.8.1	Labeled statements	146
6.8.2	Compound statement	147
6.8.3	Expression and null statements	147
6.8.4	Selection statements	148
6.8.5	Iteration statements	150
6.8.6	Jump statements	151
6.9	External definitions	155
6.9.1	Function definitions	156
6.9.2	External object definitions	158
6.10	Preprocessing directives	160
6.10.1	Conditional inclusion	162
6.10.2	Source file inclusion	164
6.10.3	Macro replacement	166

6.10.4	Line control	173
6.10.5	Error directive	174
6.10.6	Pragma directive	174
6.10.7	Null directive	175
6.10.8	Predefined macro names	175
6.10.9	Pragma operator	178
6.11	Future language directions	179
6.11.1	Floating types	179
6.11.2	Linkages of identifiers	179
6.11.3	External names	179
6.11.4	Character escape sequences	179
6.11.5	Storage-class specifiers	179
6.11.6	Function declarators	179
6.11.7	Function definitions	179
6.11.8	Pragma directives	179
6.11.9	Predefined macro names	179
7.	Library	180
7.1	Introduction	180
7.1.1	Definitions of terms	180
7.1.2	Standard headers	181
7.1.3	Reserved identifiers	182
7.1.4	Use of library functions	183
7.2	Diagnostics <assert.h>	186
7.2.1	Program diagnostics	186
7.3	Complex arithmetic <complex.h>	188
7.3.1	Introduction	188
7.3.2	Conventions	189
7.3.3	Branch cuts	189
7.3.4	The CX_LIMITED_RANGE pragma	189
7.3.5	Trigonometric functions	190
7.3.6	Hyperbolic functions	192
7.3.7	Exponential and logarithmic functions	194
7.3.8	Power and absolute-value functions	195
7.3.9	Manipulation functions	196
7.4	Character handling <ctype.h>	200
7.4.1	Character classification functions	200
7.4.2	Character case mapping functions	203
7.5	Errors <errno.h>	205
7.6	Floating-point environment <fenv.h>	206
7.6.1	The FENV_ACCESS pragma	208
7.6.2	Floating-point exceptions	209
7.6.3	Rounding	212
7.6.4	Environment	213
7.7	Characteristics of floating types <float.h>	216

7.8	Format conversion of integer types <inttypes.h>	217
7.8.1	Macros for format specifiers	217
7.8.2	Functions for greatest-width integer types	218
7.9	Alternative spellings <iso646.h>	221
7.10	Sizes of integer types <limits.h>	222
7.11	Localization <locale.h>	223
7.11.1	Locale control	224
7.11.2	Numeric formatting convention inquiry	225
7.12	Mathematics <math.h>	231
7.12.1	Treatment of error conditions	233
7.12.2	The FP_CONTRACT pragma	235
7.12.3	Classification macros	235
7.12.4	Trigonometric functions	238
7.12.5	Hyperbolic functions	240
7.12.6	Exponential and logarithmic functions	242
7.12.7	Power and absolute-value functions	247
7.12.8	Error and gamma functions	249
7.12.9	Nearest integer functions	251
7.12.10	Remainder functions	254
7.12.11	Manipulation functions	255
7.12.12	Maximum, minimum, and positive difference functions	257
7.12.13	Floating multiply-add	258
7.12.14	Comparison macros	259
7.13	Nonlocal jumps <setjmp.h>	262
7.13.1	Save calling environment	262
7.13.2	Restore calling environment	263
7.14	Signal handling <signal.h>	265
7.14.1	Specify signal handling	266
7.14.2	Send signal	267
7.15	Alignment <stdalign.h>	268
7.16	Variable arguments <stdarg.h>	269
7.16.1	Variable argument list access macros	269
7.17	Atomics <stdatomic.h>	273
7.17.1	Introduction	273
7.17.2	Initialization	274
7.17.3	Order and consistency	275
7.17.4	Fences	278
7.17.5	Lock-free property	279
7.17.6	Atomic integer types	280
7.17.7	Operations on atomic types	282
7.17.8	Atomic flag type and operations	285
7.18	Boolean type and values <stdbool.h>	287
7.19	Common definitions <stddef.h>	288
7.20	Integer types <stdint.h>	289

7.20.1	Integer types	289
7.20.2	Limits of specified-width integer types	291
7.20.3	Limits of other integer types	293
7.20.4	Macros for integer constants	294
7.21	Input/output <stdio.h>	296
7.21.1	Introduction	296
7.21.2	Streams	298
7.21.3	Files	300
7.21.4	Operations on files	302
7.21.5	File access functions	304
7.21.6	Formatted input/output functions	309
7.21.7	Character input/output functions	330
7.21.8	Direct input/output functions	335
7.21.9	File positioning functions	336
7.21.10	Error-handling functions	338
7.22	General utilities <stdlib.h>	340
7.22.1	Numeric conversion functions	341
7.22.2	Pseudo-random sequence generation functions	346
7.22.3	Memory management functions	347
7.22.4	Communication with the environment	350
7.22.5	Searching and sorting utilities	354
7.22.6	Integer arithmetic functions	356
7.22.7	Multibyte/wide character conversion functions	357
7.22.8	Multibyte/wide string conversion functions	359
7.23	_Noreturn <stdnoreturn.h>	361
7.24	String handling <string.h>	362
7.24.1	String function conventions	362
7.24.2	Copying functions	362
7.24.3	Concatenation functions	364
7.24.4	Comparison functions	365
7.24.5	Search functions	367
7.24.6	Miscellaneous functions	371
7.25	Type-generic math <tgmath.h>	373
7.26	Threads <threads.h>	376
7.26.1	Introduction	376
7.26.2	Initialization functions	378
7.26.3	Condition variable functions	378
7.26.4	Mutex functions	380
7.26.5	Thread functions	383
7.26.6	Thread-specific storage functions	386
7.27	Date and time <time.h>	388
7.27.1	Components of time	388
7.27.2	Time manipulation functions	389
7.27.3	Time conversion functions	392

7.28	Unicode utilities <uchar.h>	398
7.28.1	Restartable multibyte/wide character conversion functions	398
7.29	Extended multibyte and wide character utilities <wchar.h>	402
7.29.1	Introduction	402
7.29.2	Formatted wide character input/output functions	403
7.29.3	Wide character input/output functions	421
7.29.4	General wide string utilities	426
7.29.4.1	Wide string numeric conversion functions	426
7.29.4.2	Wide string copying functions	430
7.29.4.3	Wide string concatenation functions	432
7.29.4.4	Wide string comparison functions	433
7.29.4.5	Wide string search functions	435
7.29.4.6	Miscellaneous functions	439
7.29.5	Wide character time conversion functions	439
7.29.6	Extended multibyte/wide character conversion utilities	440
7.29.6.1	Single-byte/wide character conversion functions	441
7.29.6.2	Conversion state functions	441
7.29.6.3	Restartable multibyte/wide character conversion functions	442
7.29.6.4	Restartable multibyte/wide string conversion functions	444
7.30	Wide character classification and mapping utilities <wctype.h>	447
7.30.1	Introduction	447
7.30.2	Wide character classification utilities	448
7.30.2.1	Wide character classification functions	448
7.30.2.2	Extensible wide character classification functions	451
7.30.3	Wide character case mapping utilities	453
7.30.3.1	Wide character case mapping functions	453
7.30.3.2	Extensible wide character case mapping functions	453
7.31	Future library directions	455
7.31.1	Complex arithmetic <complex.h>	455
7.31.2	Character handling <ctype.h>	455
7.31.3	Errors <errno.h>	455
7.31.4	Floating-point environment <fenv.h>	455
7.31.5	Format conversion of integer types <inttypes.h>	455
7.31.6	Localization <locale.h>	455
7.31.7	Signal handling <signal.h>	455
7.31.8	Atomics <stdatomic.h>	455
7.31.9	Boolean type and values <stdbool.h>	456
7.31.10	Integer types <stdint.h>	456
7.31.11	Input/output <stdio.h>	456
7.31.12	General utilities <stdlib.h>	456

7.31.13	String handling <string.h>	456
7.31.14	Date and time <time.h>	456
7.31.15	Threads <threads.h>	456
7.31.16	Extended multibyte and wide character utilities <wchar.h>	456
7.31.17	Wide character classification and mapping utilities <wctype.h>	457
Annex A	(informative) Language syntax summary	458
A.1	Lexical grammar	458
A.2	Phrase structure grammar	465
A.3	Preprocessing directives	473
Annex B	(informative) Library summary	475
B.1	Diagnostics <assert.h>	475
B.2	Complex <complex.h>	475
B.3	Character handling <ctype.h>	477
B.4	Errors <errno.h>	477
B.5	Floating-point environment <fenv.h>	477
B.6	Characteristics of floating types <float.h>	478
B.7	Format conversion of integer types <inttypes.h>	478
B.8	Alternative spellings <iso646.h>	479
B.9	Sizes of integer types <limits.h>	479
B.10	Localization <locale.h>	479
B.11	Mathematics <math.h>	479
B.12	Nonlocal jumps <setjmp.h>	484
B.13	Signal handling <signal.h>	484
B.14	Alignment <stdalign.h>	485
B.15	Variable arguments <stdarg.h>	485
B.16	Atomics <stdatomic.h>	485
B.17	Boolean type and values <stdbool.h>	487
B.18	Common definitions <stddef.h>	487
B.19	Integer types <stdint.h>	487
B.20	Input/output <stdio.h>	488
B.21	General utilities <stdlib.h>	491
B.22	_Noreturn <stdnoreturn.h>	493
B.23	String handling <string.h>	493
B.24	Type-generic math <tgmath.h>	495
B.25	Threads <threads.h>	495
B.26	Date and time <time.h>	496
B.27	Unicode utilities <uchar.h>	497
B.28	Extended multibyte/wide character utilities <wchar.h>	497
B.29	Wide character classification and mapping utilities <wctype.h>	502
Annex C	(informative) Sequence points	503

Annex D (normative) Universal character names for identifiers	504
D.1 Ranges of characters allowed	504
D.2 Ranges of characters disallowed initially	504
Annex E (informative) Implementation limits	505
Annex F (normative) IEC 60559 floating-point arithmetic	507
F.1 Introduction	507
F.2 Types	507
F.3 Operators and functions	508
F.4 Floating to integer conversion	510
F.5 Binary-decimal conversion	510
F.6 The return statement	511
F.7 Contracted expressions	511
F.8 Floating-point environment	511
F.9 Optimization	514
F.10 Mathematics <math.h>	517
F.10.1 Trigonometric functions	518
F.10.2 Hyperbolic functions	520
F.10.3 Exponential and logarithmic functions	520
F.10.4 Power and absolute value functions	524
F.10.5 Error and gamma functions	525
F.10.6 Nearest integer functions	526
F.10.7 Remainder functions	528
F.10.8 Manipulation functions	529
F.10.9 Maximum, minimum, and positive difference functions	530
F.10.10 Floating multiply-add	530
F.10.11 Comparison macros	531
Annex G (normative) IEC 60559-compatible complex arithmetic	532
G.1 Introduction	532
G.2 Types	532
G.3 Conventions	532
G.4 Conversions	533
G.4.1 Imaginary types	533
G.4.2 Real and imaginary	533
G.4.3 Imaginary and complex	533
G.5 Binary operators	533
G.5.1 Multiplicative operators	534
G.5.2 Additive operators	537
G.6 Complex arithmetic <complex.h>	537
G.6.1 Trigonometric functions	539
G.6.2 Hyperbolic functions	539
G.6.3 Exponential and logarithmic functions	543
G.6.4 Power and absolute-value functions	544
G.7 Type-generic math <tgmath.h>	545

Annex H (informative) Language independent arithmetic	546
H.1 Introduction	546
H.2 Types	546
H.3 Notification	550
Annex I (informative) Common warnings	552
Annex J (informative) Portability issues	554
J.1 Unspecified behavior	554
J.2 Undefined behavior	557
J.3 Implementation-defined behavior	571
J.4 Locale-specific behavior	578
J.5 Common extensions	579
Annex K (normative) Bounds-checking interfaces	582
K.1 Background	582
K.2 Scope	583
K.3 Library	583
K.3.1 Introduction	583
K.3.1.1 Standard headers	583
K.3.1.2 Reserved identifiers	584
K.3.1.3 Use of errno	584
K.3.1.4 Runtime-constraint violations	584
K.3.2 Errors <errno.h>	585
K.3.3 Common definitions <stddef.h>	585
K.3.4 Integer types <stdint.h>	585
K.3.5 Input/output <stdio.h>	586
K.3.5.1 Operations on files	586
K.3.5.2 File access functions	588
K.3.5.3 Formatted input/output functions	591
K.3.5.4 Character input/output functions	602
K.3.6 General utilities <stdlib.h>	604
K.3.6.1 Runtime-constraint handling	604
K.3.6.2 Communication with the environment	606
K.3.6.3 Searching and sorting utilities	607
K.3.6.4 Multibyte/wide character conversion functions	610
K.3.6.5 Multibyte/wide string conversion functions	611
K.3.7 String handling <string.h>	614
K.3.7.1 Copying functions	614
K.3.7.2 Concatenation functions	617
K.3.7.3 Search functions	620
K.3.7.4 Miscellaneous functions	621
K.3.8 Date and time <time.h>	624
K.3.8.1 Components of time	624
K.3.8.2 Time conversion functions	624

K.3.9	Extended multibyte and wide character utilities	
<wchar.h>		627
K.3.9.1	Formatted wide character input/output functions	628
K.3.9.2	General wide string utilities	639
K.3.9.3	Extended multibyte/wide character conversion utilities	647
Annex L (normative)	Analyzability	652
L.1	Scope	652
L.2	Definitions	652
L.3	Requirements	653
Bibliography		654
Index		657

Foreword

- 1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are member of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.
- 2 International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.
- 3 The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.
- 4 Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.
- 5 ISO/IEC 9899 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.
- 6 This third edition cancels and replaces the second edition, ISO/IEC 9899:1999, which has been technically revised. It also incorporates the Technical Corrigenda ISO/IEC 9899:1999/Cor 1:2001, ISO/IEC 9899:1999/Cor 2:2004, and ISO/IEC 9899:1999/Cor 3:2007. Major changes from the previous edition include:
 - conditional (optional) features (including some that were previously mandatory)
 - support for multiple threads of execution including an improved memory sequencing model, atomic objects, and thread-local storage (`<stdatomic.h>` and `<threads.h>`)
 - additional floating-point characteristic macros (`<float.h>`)
 - querying and specifying alignment of objects (`<stdalign.h>`, `<stdlib.h>`)
 - Unicode characters and strings (`<uchar.h>`) (originally specified in ISO/IEC TR 19769:2004)
 - type-generic expressions
 - static assertions
 - anonymous structures and unions

- no-return functions
- macros to create complex numbers (`<complex.h>`)
- support for opening files for exclusive access
- removed the `gets` function (`<stdio.h>`)
- added the `aligned_alloc`, `at_quick_exit`, and `quick_exit` functions (`<stdlib.h>`)
- (conditional) support for bounds-checking interfaces (originally specified in ISO/IEC TR 24731-1:2007)
- (conditional) support for analyzability

7 Major changes in the second edition included:

- restricted character set support via digraphs and `<iso646.h>` (originally specified in ISO/IEC 9899:1990/Amd.1:1995)
- wide character library support in `<wchar.h>` and `<wctype.h>` (originally specified in ISO/IEC 9899:1990/Amd.1:1995)
- more precise aliasing rules via effective type
- restricted pointers
- variable length arrays
- flexible array members
- `static` and type qualifiers in parameter array declarators
- complex (and imaginary) support in `<complex.h>`
- type-generic math macros in `<tgmath.h>`
- the `long long int` type and library functions
- extended integer types
- increased minimum translation limits
- additional floating-point characteristics in `<float.h>`
- remove implicit `int`
- reliable integer division
- universal character names (`\u` and `\U`)
- extended identifiers
- hexadecimal floating-point constants and `%a` and `%A` `printf/scanf` conversion specifiers
- compound literals

- designated initializers
- `//` comments
- specified width integer types and corresponding library functions in `<inttypes.h>` and `<stdint.h>`
- remove implicit function declaration
- preprocessor arithmetic done in `intmax_t/uintmax_t`
- mixed declarations and statements
- new block scopes for selection and iteration statements
- integer constant type rules
- integer promotion rules
- macros with a variable number of arguments
- the `vscanf` family of functions in `<stdio.h>` and `<wchar.h>`
- additional math library functions in `<math.h>`
- treatment of error conditions by math library functions (`math_errhandling`)
- floating-point environment access in `<fenv.h>`
- IEC 60559 (also known as IEC 559 or IEEE arithmetic) support
- trailing comma allowed in `enum` declaration
- `%lf` conversion specifier allowed in `printf`
- inline functions
- the `snprintf` family of functions in `<stdio.h>`
- boolean type in `<stdbool.h>`
- idempotent type qualifiers
- empty macro arguments
- new structure type compatibility rules (tag compatibility)
- additional predefined macro names
- `_Pragma` preprocessing operator
- standard pragmas
- `__func__` predefined identifier
- `va_copy` macro
- additional `strftime` conversion specifiers
- LIA compatibility annex

- deprecate **ungetc** at the beginning of a binary file
- remove deprecation of aliased array parameters
- conversion of array to pointer not limited to lvalues
- relaxed constraints on aggregate and union initialization
- relaxed restrictions on portable header names
- **return** without expression not permitted in function that returns a value (and vice versa)

Introduction

- 1 With the introduction of new devices and extended character sets, new features may be added to this International Standard. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.
- 2 Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this International Standard. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.31]) is discouraged.
- 3 This International Standard is divided into four major subdivisions:
 - preliminary elements (clauses 1–4);
 - the characteristics of environments that translate and execute C programs (clause 5);
 - the language syntax, constraints, and semantics (clause 6);
 - the library facilities (clause 7).
- 4 Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this International Standard. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes provide additional information and summarize the information contained in this International Standard. A bibliography lists documents that were referred to during the preparation of the standard.
- 5 The language clause (clause 6) is derived from “The C Reference Manual”.
- 6 The library clause (clause 7) is based on the *1984 /usr/group Standard*.
- 7 The Working Group responsible for this standard (WG 14) maintains a site on the World Wide Web at <http://www.open-std.org/JTC1/SC22/WG14/> containing additional information relevant to this standard such as a Rationale for many of the decisions made during its preparation and a log of Defect Reports and Responses.

Information technology — Programming languages — C

1. Scope

- 1 This International Standard specifies the form and establishes the interpretation of programs written in the C programming language.¹⁾ It specifies
 - the representation of C programs;
 - the syntax and constraints of the C language;
 - the semantic rules for interpreting C programs;
 - 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.
- 2 This International Standard does not specify
 - the mechanism by which C programs are transformed for use by a data-processing system;
 - 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;
 - all minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

1) This International 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 programmers.

2. Normative references

- 1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.
- 2 ISO/IEC 2382-1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*.
- 3 ISO 4217, *Codes for the representation of currencies and funds*.
- 4 ISO 8601, *Data elements and interchange formats — Information interchange — Representation of dates and times*.
- 5 ISO/IEC 10646, *Information technology — Universal Coded Character Set (UCS)*.
- 6 IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989).
- 7 ISO 80000-2, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*.

3. Terms, definitions, and symbols

- 1 For the purposes of this document, the following terms and definitions apply. Other terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard are to be interpreted according to ISO/IEC 2382–1. Mathematical symbols not defined in this International Standard are to be interpreted according to ISO 80000–2.

3.1

- 1 **access**

⟨execution-time action⟩ to read or modify the value of an object

- 2 NOTE 1 Where only one of these two actions is meant, “read” or “modify” is used.
- 3 NOTE 2 “Modify” includes the case where the new value being stored is the same as the previous value.
- 4 NOTE 3 Expressions that are not evaluated do not access objects.

3.2

- 1 **alignment**

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

3.3

- 1 **argument**

actual argument

actual parameter (deprecated)

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

3.4

- 1 **behavior**

external appearance or action

3.4.1

- 1 **implementation-defined behavior**

unspecified behavior where each implementation documents how the choice is made

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

3.4.2

- 1 **locale-specific behavior**

behavior that depends on local conventions of nationality, culture, and language that each implementation documents

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

3.4.3

1 undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

- 2 NOTE Possible 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).

- 3 EXAMPLE An example of undefined behavior is the behavior on integer overflow.

3.4.4

1 unspecified behavior

use of an unspecified value, or other behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

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

3.5

1 bit

unit of data storage in the execution environment large enough to hold an object that may have one of two values

- 2 NOTE It need not be possible to express the address of each individual bit of an object.

3.6

1 byte

addressable unit of data storage large enough to hold any member of the basic character set of the execution environment

- 2 NOTE 1 It is possible to express the address of each individual byte of an object uniquely.

- 3 NOTE 2 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*.

3.7

1 character

⟨abstract⟩ member of a set of elements used for the organization, control, or representation of data

3.7.1

1 character

single-byte character

⟨C⟩ bit representation that fits in a byte

3.7.2

1 **multibyte character**

sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment

2 NOTE The extended character set is a superset of the basic character set.

3.7.3

1 **wide character**

value representable by an object of type **wchar_t**, capable of representing any character in the current locale

3.8

1 **constraint**

restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted

3.9

1 **correctly rounded result**

representation in the result format that is nearest in value, subject to the current rounding mode, to what the result would be given unlimited range and precision

3.10

1 **diagnostic message**

message belonging to an implementation-defined subset of the implementation's message output

3.11

1 **forward reference**

reference to a later subclause of this International Standard that contains additional information relevant to this subclause

3.12

1 **implementation**

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

3.13

1 **implementation limit**

restriction imposed upon programs by the implementation

3.14

1 **memory location**

either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width

2 NOTE 1 Two threads of execution can update and access separate memory locations without interfering with each other.

3 NOTE 2 A bit-field and an adjacent non-bit-field member are in separate memory locations. The same applies to two bit-fields, if one is declared inside a nested structure declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field member declaration. It is not safe to concurrently update two non-atomic bit-fields in the same structure if all members declared between them are also (non-zero-length) bit-fields, no matter what the sizes of those intervening bit-fields happen to be.

4 EXAMPLE A structure declared as

```
struct {
    char a;
    int b:5, c:11, :0, d:8;
    struct { int ee:8; } e;
}
```

contains four separate memory locations: The member **a**, and bit-fields **d** and **e.ee** are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields **b** and **c** together constitute the fourth memory location. The bit-fields **b** and **c** cannot be concurrently modified, but **b** and **a**, for example, can be.

3.15

1 object

region of data storage in the execution environment, the contents of which can represent values

2 NOTE When referenced, an object may be interpreted as having a particular type; see 6.3.2.1.

3.16

1 parameter

formal parameter

formal argument (deprecated)

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

3.17

1 recommended practice

specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations

3.18

1 runtime-constraint

requirement on a program when calling a library function

2 NOTE 1 Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by 3.8, and need not be diagnosed at translation time.

3 NOTE 2 Implementations that support the extensions in annex K are required to verify that the runtime-constraints for a library function are not violated by the program; see K.3.1.4.

3.19**1 value**

precise meaning of the contents of an object when interpreted as having a specific type

3.19.1**1 implementation-defined value**

unspecified value where each implementation documents how the choice is made

3.19.2**1 indeterminate value**

either an unspecified value or a trap representation

3.19.3**1 unspecified value**

valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance

2 NOTE An unspecified value cannot be a trap representation.

3.19.4**1 trap representation**

an object representation that need not represent a value of the object type

3.19.5**1 perform a trap**

interrupt execution of the program such that no further operations are performed

2 NOTE In this International Standard, when the word “trap” is not immediately followed by “representation”, this is the intended usage.²⁾

3.20**1 $\lceil x \rceil$**

ceiling of x : the least integer greater than or equal to x

2 EXAMPLE $\lceil 2.4 \rceil$ is 3, $\lceil -2.4 \rceil$ is -2.

3.21**1 $\lfloor x \rfloor$**

floor of x : the greatest integer less than or equal to x

2 EXAMPLE $\lfloor 2.4 \rfloor$ is 2, $\lfloor -2.4 \rfloor$ is -3.

2) For example, “Trapping or stopping (if supported) is disabled...” (F.8.2). Note that fetching a trap representation might perform a trap but is not required to (see 6.2.6.1).

4. Conformance

- 1 In this International 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.
- 2 If a “shall” or “shall not” requirement that appears outside of a constraint or runtime-constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International 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”.
- 3 A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3.
- 4 The implementation shall not successfully translate a preprocessing translation unit containing a **#error** preprocessing directive unless it is part of a group skipped by conditional inclusion.
- 5 A *strictly conforming program* shall use only those features of the language and library specified in this International Standard.³⁾ It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.
- 6 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 clause (clause 7) is confined to the contents of the standard headers `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdalign.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, and `<stdnoreturn.h>`. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.⁴⁾

-
- 3) A strictly conforming program can use conditional features (see 6.10.8.3) provided the use is guarded by an appropriate conditional inclusion preprocessing directive using the related macro. For example:

```
#ifdef __STDC_IEC_559__ /* FE_UPWARD defined */
    /* ... */
    fesetround(FE_UPWARD);
    /* ... */
#endif
```

- 4) This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this International Standard.

- 7 A *conforming program* is one that is acceptable to a conforming implementation.⁵⁾
- 8 An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

Forward references: conditional inclusion (6.10.1), error directive (6.10.5), characteristics of floating types `<float.h>` (7.7), alternative spellings `<iso646.h>` (7.9), sizes of integer types `<limits.h>` (7.10), alignment `<stdalign.h>` (7.15), variable arguments `<stdarg.h>` (7.16), boolean type and values `<stdbool.h>` (7.18), common definitions `<stddef.h>` (7.19), integer types `<stdint.h>` (7.20), `<stdnoreturn.h>` (7.23).

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

5. Environment

- 1 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 International Standard. Their characteristics define and constrain the results of executing conforming C programs constructed according to the syntactic and semantic rules for conforming implementations.

Forward references: In this clause, only a few of many possible forward references have been noted.

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Program structure

- 1 A C program need not all be translated at the same time. The text of the program is kept in units called *source files*, (or *preprocessing files*) in this International Standard. A source file together with all the headers and source files included via the preprocessing directive **#include** is known as a *preprocessing translation unit*. After preprocessing, a preprocessing translation unit 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 executable program.

Forward references: linkages of identifiers (6.2.2), external definitions (6.9), preprocessing directives (6.10).

5.1.1.2 Translation phases

- 1 The precedence among the syntax rules of translation is specified by the following phases.⁶⁾
 1. Physical source file multibyte characters are mapped, in an implementation-defined manner, 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.

6) Implementations shall behave as if these separate phases occur, even though many are typically folded together in practice. Source files, translation units, and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation.

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place.
3. The source file is decomposed into preprocessing tokens⁷⁾ and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. Each comment is replaced by one space character. New-line characters 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, macro invocations are expanded, and **_Pragma** unary operator expressions are executed. If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined. A **#include** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.
5. Each source character set member and escape sequence in character constants and string literals is converted to the corresponding member of the execution character set; if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.⁸⁾
6. Adjacent 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 as a translation unit.
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: universal character names (6.4.3), lexical elements (6.4), preprocessing directives (6.10), trigraph sequences (5.2.1.1), external definitions (6.9).

7) As described in 6.4, 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.

8) An implementation need not convert all non-corresponding source characters to the same execution character.

5.1.1.3 Diagnostics

- 1 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. Diagnostic messages need not be produced in other circumstances.⁹⁾

- 2 EXAMPLE An implementation is required to issue a diagnostic for the translation unit:

```
char i;  
int i;
```

because in those cases where wording in this International Standard describes the behavior for a construct as being both a constraint error and resulting in undefined behavior, the constraint error shall be diagnosed.

5.1.2 Execution environments

- 1 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 with static storage duration 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: storage durations of objects (6.2.4), initialization (6.7.9).

5.1.2.1 Freestanding environment

- 1 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. Any library facilities available to a freestanding program, other than the minimal set required by clause 4, are implementation-defined.
- 2 The effect of program termination in a freestanding environment is implementation-defined.

5.1.2.2 Hosted environment

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

9) 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. It may also successfully translate an invalid program.

5.1.2.2.1 Program startup

- 1 The function called at program startup is named **main**. The implementation declares no prototype for this function. It shall be defined with a return type of **int** and 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):

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

or equivalent;¹⁰⁾ or in some other implementation-defined manner.

- 2 If they are declared, 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.
 - 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 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 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.

5.1.2.2.2 Program execution

- 1 In a hosted environment, a program may use all the functions, macros, type definitions, and objects described in the library clause (clause 7).

10) Thus, **int** can be replaced by a typedef name defined as **int**, or the type of **argv** can be written as **char ** argv**, and so on.

5.1.2.2.3 Program termination

- 1 If the return type of the **main** function is a type compatible with **int**, 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;¹¹⁾ reaching the **}** that terminates the **main** function returns a value of 0. If the return type is not compatible with **int**, the termination status returned to the host environment is unspecified.

Forward references: definition of terms (7.1.1), the **exit** function (7.22.4.4).

5.1.2.3 Program execution

- 1 The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.
- 2 Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*,¹²⁾ which are changes in the state of the execution environment. *Evaluation* of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object.
- 3 *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations *A* and *B*, if *A* is sequenced before *B*, then the execution of *A* shall precede the execution of *B*. (Conversely, if *A* is sequenced before *B*, then *B* is *sequenced after A*.) If *A* is not sequenced before or after *B*, then *A* and *B* are *unsequenced*. Evaluations *A* and *B* are *indeterminately sequenced* when *A* is sequenced either before or after *B*, but it is unspecified which.¹³⁾ The presence of a *sequence point* between the evaluation of expressions *A* and *B* implies that every value computation and side effect associated with *A* is sequenced before every value computation and side effect associated with *B*. (A summary of the sequence points is given in annex C.)
- 4 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

11) In accordance with 6.2.4, the lifetimes of objects with automatic storage duration declared in **main** will have ended in the former case, even where they would not have in the latter.

12) The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes. Floating-point operations implicitly set the status flags; modes affect result values of floating-point operations. Implementations that support such floating-point state are required to regard changes to it as side effects — see annex F for details. The floating-point environment library **<fenv.h>** provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases.

13) The executions of unsequenced evaluations can interleave. Indeterminately sequenced evaluations cannot interleave, but can be executed in any order.

calling a function or accessing a volatile object).

- 5 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects that are neither lock-free atomic objects nor of type **volatile sig_atomic_t** are unspecified, as is the state of the floating-point environment. The value of any object modified by the handler that is neither a lock-free atomic object nor of type **volatile sig_atomic_t** becomes indeterminate when the handler exits, as does the state of the floating-point environment if it is modified by the handler and not restored to its original state.
- 6 The least requirements on a conforming implementation are:
 - Accesses to volatile objects are evaluated strictly according to the rules of the abstract machine.
 - 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 7.21.3. 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.

This is the *observable behavior* of the program.

- 7 What constitutes an interactive device is implementation-defined.
- 8 More stringent correspondences between abstract and actual semantics may be defined by each implementation.
- 9 EXAMPLE 1 An implementation might define a one-to-one correspondence between abstract and actual 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.
- 10 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 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 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.
- 11 EXAMPLE 2 In executing the fragment

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

the “integer promotions” require that the abstract machine promote the value of each variable to **int** size and then add the two **ints** and truncate the sum. Provided the addition of two **chars** can be done without

overflow, or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

- 12 EXAMPLE 3 Similarly, in the fragment

```
float f1, f2;
double d;
/* ... */
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**).

- 13 EXAMPLE 4 Implementations employing wide registers have to take care to honor appropriate semantics. Values are independent of whether they are represented in a register or in memory. For example, an implicit *spilling* of a register is not permitted to alter the value. Also, an explicit *store and load* is required to round to the precision of the storage type. In particular, casts and assignments are required to perform their specified conversion. For the fragment

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

the values assigned to **d1** and **d2** are required to have been converted to **float**.

- 14 EXAMPLE 5 Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real numbers are often not valid (see F.9).

```
double x, y, z;
/* ... */
x = (x * y) * z; // not equivalent to x *= y * z;
z = (x - y) + y; // not equivalent to z = x;
z = x + x * y;   // not equivalent to z = x * (1.0 + y);
y = x / 5.0;     // not equivalent to y = x * 0.2;
```

- 15 EXAMPLE 6 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

```
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 explicit trap and in which the range of values representable by an **int** is $[-32768, +32767]$, the implementation cannot rewrite this 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 a trap

while the original expression would not; nor can the expression be rewritten either as

```
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 overflow silently generates some value and where positive and negative overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

- 16 **EXAMPLE 7** The grouping of an expression does not completely determine its evaluation. In the following fragment

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

the expression statement is grouped as if it were written as

```
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: expressions (6.5), type qualifiers (6.7.3), statements (6.8), floating-point environment **<fenv.h>** (7.6), the **signal** function (7.14), files (7.21.3).

5.1.2.4 Multi-threaded executions and data races

- 1 Under a hosted implementation, a program can have more than one *thread of execution* (or *thread*) running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads.¹⁴⁾ Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.
- 2 The value of an object visible to a thread *T* at a particular point is the initial value of the object, a value stored in the object by *T*, or a value stored in the object by another thread, according to the rules below.
- 3 **NOTE 1** In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs.
- 4 Two expression evaluations *conflict* if one of them modifies a memory location and the other one reads or modifies the same memory location.

14) The execution can usually be viewed as an interleaving of all of the threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving as described below.

- 5 The library defines a number of *atomic operations* (7.17) and operations on mutexes (7.26.4) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A *synchronization operation* on one or more memory locations is either an *acquire operation*, a *release operation*, both an acquire and release operation, or a *consume operation*. A synchronization operation without an associated memory location is a *fence* and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are *relaxed atomic operations*, which are not synchronization operations, and *atomic read-modify-write operations*, which have special characteristics.
- 6 NOTE 2 For example, a call that acquires a mutex will perform an acquire operation on the locations composing the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on *A* forces prior side effects on other memory locations to become visible to other threads that later perform an acquire or consume operation on *A*. We do not include relaxed atomic operations as synchronization operations although, like synchronization operations, they cannot contribute to data races.
- 7 All modifications to a particular atomic object *M* occur in some particular total order, called the *modification order* of *M*. If *A* and *B* are modifications of an atomic object *M*, and *A* happens before *B*, then *A* shall precede *B* in the modification order of *M*, which is defined below.
- 8 NOTE 3 This states that the modification orders must respect the “happens before” relation.
- 9 NOTE 4 There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads may observe modifications to different variables in inconsistent orders.
- 10 A *release sequence* headed by a release operation *A* on an atomic object *M* is a maximal contiguous sub-sequence of side effects in the modification order of *M*, where the first operation is *A* and every subsequent operation either is performed by the same thread that performed the release or is an atomic read-modify-write operation.
- 11 Certain library calls *synchronize with* other library calls performed by another thread. In particular, an atomic operation *A* that performs a release operation on an object *M* synchronizes with an atomic operation *B* that performs an acquire operation on *M* and reads a value written by any side effect in the release sequence headed by *A*.
- 12 NOTE 5 Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation.
- 13 NOTE 6 The specifications of the synchronization operations define when one reads the value written by another. For atomic variables, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition “reads the value written” by the last mutex release.
- 14 An evaluation *A* carries a dependency¹⁵⁾ to an evaluation *B* if:

15) The “carries a dependency” relation is a subset of the “sequenced before” relation, and is similarly strictly intra-thread.

- the value of *A* is used as an operand of *B*, unless:
 - *B* is an invocation of the **kill_dependency** macro,
 - *A* is the left operand of a **&&** or **| |** operator,
 - *A* is the left operand of a **? :** operator, or
 - *A* is the left operand of a **,** operator;
 or
 - *A* writes a scalar object or bit-field *M*, *B* reads from *M* the value written by *A*, and *A* is sequenced before *B*, or
 - for some evaluation *X*, *A* carries a dependency to *X* and *X* carries a dependency to *B*.
- 15 An evaluation *A* is *dependency-ordered before*¹⁶⁾ an evaluation *B* if:
- *A* performs a release operation on an atomic object *M*, and, in another thread, *B* performs a consume operation on *M* and reads a value written by any side effect in the release sequence headed by *A*, or
 - for some evaluation *X*, *A* is dependency-ordered before *X* and *X* carries a dependency to *B*.
- 16 An evaluation *A* *inter-thread happens before* an evaluation *B* if *A* synchronizes with *B*, *A* is dependency-ordered before *B*, or, for some evaluation *X*:
- *A* synchronizes with *X* and *X* is sequenced before *B*,
 - *A* is sequenced before *X* and *X* inter-thread happens before *B*, or
 - *A* inter-thread happens before *X* and *X* inter-thread happens before *B*.
- 17 NOTE 7 The “inter-thread happens before” relation describes arbitrary concatenations of “sequenced before”, “synchronizes with”, and “dependency-ordered before” relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with “dependency-ordered before” followed by “sequenced before”. The reason for this limitation is that a consume operation participating in a “dependency-ordered before” relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of “sequenced before”. The reasons for this limitation are (1) to permit “inter-thread happens before” to be transitively closed and (2) the “happens before” relation, defined below, provides for relationships consisting entirely of “sequenced before”.
- 18 An evaluation *A* *happens before* an evaluation *B* if *A* is sequenced before *B* or *A* inter-thread happens before *B*.

16) The “dependency-ordered before” relation is analogous to the “synchronizes with” relation, but uses release/consume in place of release/acquire.

- 19 A *visible side effect* A on an object M with respect to a value computation B of M satisfies the conditions:

- A happens before B , and
- there is no other side effect X to M such that A happens before X and X happens before B .

The value of a non-atomic scalar object M , as determined by evaluation B , shall be the value stored by the visible side effect A .

- 20 NOTE 8 If there is ambiguity about which side effect to a non-atomic object is visible, then there is a data race and the behavior is undefined.
- 21 NOTE 9 This states that operations on ordinary variables are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined here, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution.
- 22 The *visible sequence of side effects* on an atomic object M , with respect to a value computation B of M , is a maximal contiguous sub-sequence of side effects in the modification order of M , where the first side effect is visible with respect to B , and for every subsequent side effect, it is not the case that B happens before it. The value of an atomic object M , as determined by evaluation B , shall be the value stored by some operation in the visible sequence of M with respect to B . Furthermore, if a value computation A of an atomic object M happens before a value computation B of M , and the value computed by A corresponds to the value stored by side effect X , then the value computed by B shall either equal the value computed by A , or be the value stored by side effect Y , where Y follows X in the modification order of M .
- 23 NOTE 10 This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are “relaxed” loads. By doing so, we effectively make the “cache coherence” guarantee provided by most hardware available to C atomic operations.
- 24 NOTE 11 The visible sequence depends on the “happens before” relation, which in turn depends on the values observed by loads of atomics, which we are restricting here. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the “happens before” relation derived as described above, satisfy the resulting constraints as imposed here.
- 25 The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.
- 26 NOTE 12 It can be shown that programs that correctly use simple mutexes and `memory_order_seq_cst` operations to prevent all data races, and use no other synchronization operations, behave as though the operations executed by their constituent threads were simply interleaved, with each value computation of an object being the last value stored in that interleaving. This is normally referred to as “sequential consistency”. However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must contain undefined behavior.

- 27 NOTE 13 Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this standard, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. We also generally preclude reordering of atomic loads in cases in which the atomics in question may alias, since this may violate the "visible sequence" rules.
- 28 NOTE 14 Transformations that introduce a speculative read of a potentially shared memory location may not preserve the semantics of the program as defined in this standard, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection.

5.2 Environmental considerations

5.2.1 Character sets

- 1 Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written (the *source character set*), and the set interpreted in the execution environment (the *execution character set*). Each set is further divided into a *basic character set*, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called *extended characters*. The combined set is also called the *extended character set*. The values of the members of the execution character set are implementation-defined.
- 2 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, called the *null character*, shall exist in the basic execution character set; it is used to terminate a character string.
- 3 Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the Latin 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 26 *lowercase letters* of the Latin 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*

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. The representation of each member of the source and execution basic character sets shall fit in a byte. 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 International Standard treats such an end-of-line indicator as if it were a single new-line character. In the basic execution character set, there shall be control characters representing alert, backspace, carriage return, and new line. If any other characters are encountered in a source file (except in an identifier, 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.

- 4 A *letter* is an uppercase letter or a lowercase letter as defined above; in this International Standard the term does not include other characters that are letters in other alphabets.
- 5 The universal character name construct provides a way to name other characters.

Forward references: universal character names (6.4.3), character constants (6.4.4.4), preprocessing directives (6.10), string literals (6.4.5), comments (6.4.9), string (7.1.1).

5.2.1.1 Trigraph sequences

- 1 Before any other processing takes place, each occurrence of one of the following sequences of three characters (called *trigraph sequences*¹⁷⁾) is replaced with the corresponding single character.

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

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

- 2 EXAMPLE 1

```
??=define arraycheck(a, b) a??(b??) ??!??! b??(a??)
```

becomes

```
#define arraycheck(a, b) a[b] || b[a]
```

- 3 EXAMPLE 2 The following source line

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

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

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

5.2.1.2 Multibyte characters

- 1 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 need not have the same encoding as for the source character set. For both character sets, the following shall hold:
 - The basic character set shall be present and each character shall be encoded as a single byte.
 - The presence, meaning, and representation of any additional members is locale-specific.

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

- A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *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.
 - A byte with all bits zero shall be interpreted as a null character independent of shift state. Such a byte shall not occur as part of any other multibyte character.
- 2 For source files, the following shall hold:
- An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state.
 - An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

5.2.2 Character display semantics

- 1 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 printing character (as defined by the `isprint` function) to a display device is to display a graphic representation of that character at 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 of the display device is unspecified.
- 2 Alphabetic escape sequences representing nongraphic characters in the execution character set are intended to produce actions on display devices as follows:
- `\a` (*alert*) Produces an audible or visible alert without changing the active position.
 - `\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 of the display device is unspecified.
 - `\f` (*form feed*) Moves the active position to the initial position at the start of the next logical page.
 - `\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 of the display device is unspecified.
 - `\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 of the display device is unspecified.

- 3 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 identical to the internal representations, and are outside the scope of this International Standard.

Forward references: the **isprint** function (7.4.1.8), the **fputc** function (7.21.7.3).

5.2.3 Signals and interrupts

- 1 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, 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 compose the executable representation of a function) on a per-invocation basis.

5.2.4 Environmental limits

- 1 Both the translation and execution environments constrain the implementation of language translators and libraries. The following summarizes the language-related environmental limits on a conforming implementation; the library-related limits are discussed in clause 7.

5.2.4.1 Translation limits

- 1 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:¹⁸⁾
 - 127 nesting levels of blocks
 - 63 nesting levels of conditional inclusion
 - 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or **void** type in a declaration
 - 63 nesting levels of parenthesized declarators within a full declarator
 - 63 nesting levels of parenthesized expressions within a full expression
 - 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character)
 - 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each

18) Implementations should avoid imposing fixed translation limits whenever possible.

universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)¹⁹⁾

- 4095 external identifiers in one translation unit
- 511 identifiers with block scope declared in one block
- 4095 macro identifiers simultaneously defined in one preprocessing translation unit
- 127 parameters in one function definition
- 127 arguments in one function call
- 127 parameters in one macro definition
- 127 arguments in one macro invocation
- 4095 characters in a logical source line
- 4095 characters in a string literal (after concatenation)
- 65535 bytes in an object (in a hosted environment only)
- 15 nesting levels for **#included** files
- 1023 **case** labels for a **switch** statement (excluding those for any nested **switch** statements)
- 1023 members in a single structure or union
- 1023 enumeration constants in a single enumeration
- 63 levels of nested structure or union definitions in a single struct-declaration-list

5.2.4.2 Numerical limits

- 1 An implementation is required to document all the limits specified in this subclause, which are specified in the headers **<limits.h>** and **<float.h>**. Additional limits are specified in **<stdint.h>**.

Forward references: integer types **<stdint.h>** (7.20).

5.2.4.2.1 Sizes of integer types **<limits.h>**

- 1 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 integer promotions. Their implementation-defined values shall be equal or greater in magnitude

19) See “future language directions” (6.11.3).

(absolute value) to those shown, with the 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 // $-(2^7 - 1)$
- maximum value for an object of type **signed char**
SCHAR_MAX +127 // $2^7 - 1$
- maximum value for an object of type **unsigned char**
UCHAR_MAX 255 // $2^8 - 1$
- minimum value for an object of type **char**
CHAR_MIN *see below*
- 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**
SHRT_MIN -32767 // $-(2^{15} - 1)$
- maximum value for an object of type **short int**
SHRT_MAX +32767 // $2^{15} - 1$
- maximum value for an object of type **unsigned short int**
USHRT_MAX 65535 // $2^{16} - 1$
- minimum value for an object of type **int**
INT_MIN -32767 // $-(2^{15} - 1)$
- maximum value for an object of type **int**
INT_MAX +32767 // $2^{15} - 1$
- maximum value for an object of type **unsigned int**
UINT_MAX 65535 // $2^{16} - 1$
- minimum value for an object of type **long int**
LONG_MIN -2147483647 // $-(2^{31} - 1)$
- maximum value for an object of type **long int**
LONG_MAX +2147483647 // $2^{31} - 1$
- maximum value for an object of type **unsigned long int**
ULONG_MAX 4294967295 // $2^{32} - 1$

- minimum value for an object of type **long long int**
LLONG_MIN **-9223372036854775807** // $-(2^{63} - 1)$
- maximum value for an object of type **long long int**
LLONG_MAX **+9223372036854775807** // $2^{63} - 1$
- maximum value for an object of type **unsigned long long int**
ULLONG_MAX **18446744073709551615** // $2^{64} - 1$

- 2 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 be 0 and the value of **CHAR_MAX** shall be the same as that of **UCHAR_MAX**.²⁰⁾ The value **UCHAR_MAX** shall equal $2^{\text{CHAR_BIT}} - 1$.

Forward references: representations of types (6.2.6), conditional inclusion (6.10.1).

5.2.4.2.2 Characteristics of floating types <float.h>

- 1 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.²¹⁾ The following parameters are used to define the model for each floating-point type:

- s* sign (± 1)
- b* base or radix of exponent representation (an integer > 1)
- e* exponent (an integer between a minimum $e_{\min}$ and a maximum $e_{\max}$)
- p* precision (the number of base-*b* digits in the significand)
- f_k* nonnegative integers less than *b* (the significand digits)

- 2 A *floating-point number* (*x*) is defined by the following model:

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

- 3 In addition to normalized floating-point numbers ($f_1 > 0$ if $x \neq 0$), floating types may be able to contain other kinds of floating-point numbers, such as *subnormal floating-point numbers* ($x \neq 0$, $e = e_{\min}$, $f_1 = 0$) and *unnormalized floating-point numbers* ($x \neq 0$, $e > e_{\min}$, $f_1 = 0$), and values that are not floating-point numbers, such as infinities and NaNs. A *NaN* is an encoding signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception; a *signaling NaN* generally raises a floating-point exception when occurring as an

²⁰⁾ See 6.2.5.

²¹⁾ 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.

arithmetic operand.²²⁾

- 4 An implementation may give zero and values that are not floating-point numbers (such as infinities and NaNs) a sign or may leave them unsigned. Wherever such values are unsigned, any requirement in this International Standard to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored.
- 5 The minimum range of representable values for a floating type is the most negative finite floating-point number representable in that type through the most positive finite floating-point number representable in that type. In addition, if negative infinity is representable in a type, the range of that type is extended to all negative real numbers; likewise, if positive infinity is representable in a type, the range of that type is extended to all positive real numbers.
- 6 The accuracy of the floating-point operations (+, -, *, /) and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results is implementation-defined, as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>`. The implementation may state that the accuracy is unknown.
- 7 All integer values in the `<float.h>` header, except `FLT_ROUNDS`, shall be constant expressions suitable for use in `#if` preprocessing directives; all floating values shall be constant expressions. All except `DECIMAL_DIG`, `FLT_EVAL_METHOD`, `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_EVAL_METHOD` and `FLT_ROUNDS`.
- 8 The rounding mode for floating-point addition is characterized by the implementation-defined value of `FLT_ROUNDS`:²³⁾
 - 1 indeterminate
 - 0 toward zero
 - 1 to nearest
 - 2 toward positive infinity
 - 3 toward negative infinity

All other values for `FLT_ROUNDS` characterize implementation-defined rounding behavior.

22) IEC 60559:1989 specifies quiet and signaling NaNs. For implementations that do not support IEC 60559:1989, the terms quiet NaN and signaling NaN are intended to apply to encodings with similar behavior.

23) Evaluation of `FLT_ROUNDS` correctly reflects any execution-time change of rounding mode through the function `fesetround` in `<fenv.h>`.

- 9 Except for assignment and cast (which remove all extra range and precision), the values yielded by operators with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the implementation-defined value of **FLT_EVAL_METHOD**:²⁴⁾

- 1 indeterminate;
- 0 evaluate all operations and constants just to the range and precision of the type;
- 1 evaluate operations and constants of type **float** and **double** to the range and precision of the **double** type, evaluate **long double** operations and constants to the range and precision of the **long double** type;
- 2 evaluate all operations and constants to the range and precision of the **long double** type.

All other negative values for **FLT_EVAL_METHOD** characterize implementation-defined behavior.

- 10 The presence or absence of subnormal numbers is characterized by the implementation-defined values of **FLT_HAS_SUBNORM**, **DBL_HAS_SUBNORM**, and **LDBL_HAS_SUBNORM**:

- 1 indeterminate²⁵⁾
- 0 absent²⁶⁾ (type does not support subnormal numbers)
- 1 present (type does support subnormal numbers)

- 11 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater or equal in magnitude (absolute value) to those shown, with the same sign:

— radix of exponent representation, *b*

FLT_RADIX 2

24) The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. For example, if **FLT_EVAL_METHOD** is 1, then the product of two **float _Complex** operands is represented in the **double _Complex** format, and its parts are evaluated to **double**.

25) Characterization as indeterminate is intended if floating-point operations do not consistently interpret subnormal representations as zero, nor as nonzero.

26) Characterization as absent is intended if no floating-point operations produce subnormal results from non-subnormal inputs, even if the type format includes representations of subnormal numbers.