INTERNATIONAL STANDARD

ISO/IEC 9899

Fourth edition
2018-07

Information technology — Programming languages — C

Technologies de l'information — Langages de programmation — C

Reference number
ISO/IEC 9899:2018(E)

Documents
Document Sharing Hub
© ISO/IEC 2018

Full document details: ISO IEC 09899-2018

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2018

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

© ISO/IEC 2018 – All rights reserved

Documents
Document Sharing Hub

Full document details: ISO IEC 09899-2018

Contents

Foreword xi

Introduction xii

1 Scope 1

2 Normative references 2

3 Terms, definitions and symbols 3

4 Conformance 8

5 Environment 9
- 5.1 Conceptual models . . . 9
  - 5.1.1 Translation environment . . . 9
  - 5.1.2 Execution environments . . . 10
- 5.2 Environmental considerations . . . 17
  - 5.2.1 Character sets . . . 17
  - 5.2.2 Character display semantics . . . 19
  - 5.2.3 Signals and interrupts . . . 19
  - 5.2.4 Environmental limits . . . 19

6 Language 28
- 6.1 Notation . . . 28
- 6.2 Concepts . . . 28
  - 6.2.1 Scopes of identifiers . . . 28
  - 6.2.2 Linkages of identifiers . . . 29
  - 6.2.3 Name spaces of identifiers . . . 29
  - 6.2.4 Storage durations of objects . . . 30
  - 6.2.5 Types . . . 31
  - 6.2.6 Representations of types . . . 33
  - 6.2.7 Compatible type and composite type . . . 35
  - 6.2.8 Alignment of objects . . . 36
- 6.3 Conversions . . . 37
  - 6.3.1 Arithmetic operands . . . 37
  - 6.3.2 Other operands . . . 40
- 6.4 Lexical elements . . . 41
  - 6.4.1 Keywords . . . 42
  - 6.4.2 Identifiers . . . 43

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

6.4.3 Universal character names . . . 44
6.4.4 Constants . . . 45
6.4.5 String literals . . . 50
6.4.6 Punctuators . . . 52
6.4.7 Header names . . . 53
6.4.8 Preprocessing numbers . . . 53
6.4.9 Comments . . . 54
6.5 Expressions . . . 55
6.5.1 Primary expressions . . . 56
6.5.2 Postfix operators . . . 57
6.5.3 Unary operators . . . 63
6.5.4 Cast operators . . . 65
6.5.5 Multiplicative operators . . . 66
6.5.6 Additive operators . . . 66
6.5.7 Bitwise shift operators . . . 68
6.5.8 Relational operators . . . 68
6.5.9 Equality operators . . . 69
6.5.10 Bitwise AND operator . . . 70
6.5.11 Bitwise exclusive OR operator . . . 70
6.5.12 Bitwise inclusive OR operator . . . 70
6.5.13 Logical AND operator . . . 71
6.5.14 Logical OR operator . . . 71
6.5.15 Conditional operator . . . 71
6.5.16 Assignment operators . . . 72
6.5.17 Comma operator . . . 75
6.6 Constant expressions . . . 76
6.7 Declarations . . . 78
6.7.1 Storage-class specifiers . . . 79
6.7.2 Type specifiers . . . 79
6.7.3 Type qualifiers . . . 87
6.7.4 Function specifiers . . . 90
6.7.5 Alignment specifier . . . 92
6.7.6 Declarators . . . 92
6.7.7 Type names . . . 98
6.7.8 Type definitions . . . 99
6.7.9 Initialization . . . 100
6.7.10 Static assertions . . . 105
6.8 Statements and blocks . . . 106
6.8.1 Labeled statements . . . 106
6.8.2 Compound statement . . . 107

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

6.8.3 Expression and null statements . . . 107
6.8.4 Selection statements . . . 108
6.8.5 Iteration statements . . . 109
6.8.6 Jump statements . . . 110
6.9 External definitions . . . 113
6.9.1 Function definitions . . . 113
6.9.2 External object definitions . . . 115
6.10 Preprocessing directives . . . 117
6.10.1 Conditional inclusion . . . 118
6.10.2 Source file inclusion . . . 119
6.10.3 Macro replacement . . . 121
6.10.4 Line control . . . 126
6.10.5 Error directive . . . 126
6.10.6 Pragma directive . . . 127
6.10.7 Null directive . . . 127
6.10.8 Predefined macro names . . . 127
6.10.9 Pragma operator . . . 129
6.11 Future language directions . . . 130
6.11.1 Floating types . . . 130
6.11.2 Linkages of identifiers . . . 130
6.11.3 External names . . . 130
6.11.4 Character escape sequences . . . 130
6.11.5 Storage-class specifiers . . . 130
6.11.6 Function declarators . . . 130
6.11.7 Function definitions . . . 130
6.11.8 Pragma directives . . . 130
6.11.9 Predefined macro names . . . 130

7 Library **131**
7.1 Introduction . . . 131
7.1.1 Definitions of terms . . . 131
7.1.2 Standard headers . . . 131
7.1.3 Reserved identifiers . . . 132
7.1.4 Use of library functions . . . 132
7.2 Diagnostics `<assert.h>` . . . 135
7.2.1 Program diagnostics . . . 135
7.3 Complex arithmetic `<complex.h>` . . . 136
7.3.1 Introduction . . . 136
7.3.2 Conventions . . . 136
7.3.3 Branch cuts . . . 136

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

7.3.4 The **CX_LIMITED_RANGE** pragma . . . 137
7.3.5 Trigonometric functions . . . 137
7.3.6 Hyperbolic functions . . . 139
7.3.7 Exponential and logarithmic functions . . . 140
7.3.8 Power and absolute-value functions . . . 141
7.3.9 Manipulation functions . . . 142
7.4 Character handling `<ctype.h>` . . . 145
7.4.1 Character classification functions . . . 145
7.4.2 Character case mapping functions . . . 147
7.5 Errors `<errno.h>` . . . 149
7.6 Floating-point environment `<fenv.h>` . . . 150
7.6.1 The **FENV_ACCESS** pragma . . . 151
7.6.2 Floating-point exceptions . . . 152
7.6.3 Rounding . . . 154
7.6.4 Environment . . . 155
7.7 Characteristics of floating types `<float.h>` . . . 157
7.8 Format conversion of integer types `<inttypes.h>` . . . 158
7.8.1 Macros for format specifiers . . . 158
7.8.2 Functions for greatest-width integer types . . . 159
7.9 Alternative spellings `<iso646.h>` . . . 161
7.10 Sizes of integer types `<limits.h>` . . . 162
7.11 Localization `<locale.h>` . . . 163
7.11.1 Locale control . . . 163
7.11.2 Numeric formatting convention inquiry . . . 164
7.12 Mathematics `<math.h>` . . . 169
7.12.1 Treatment of error conditions . . . 170
7.12.2 The **FP_CONTRACT** pragma . . . 171
7.12.3 Classification macros . . . 172
7.12.4 Trigonometric functions . . . 173
7.12.5 Hyperbolic functions . . . 175
7.12.6 Exponential and logarithmic functions . . . 177
7.12.7 Power and absolute-value functions . . . 180
7.12.8 Error and gamma functions . . . 182
7.12.9 Nearest integer functions . . . 183
7.12.10 Remainder functions . . . 185
7.12.11 Manipulation functions . . . 186
7.12.12 Maximum, minimum, and positive difference functions . . . 187
7.12.13 Floating multiply-add . . . 188
7.12.14 Comparison macros . . . 189
7.13 Nonlocal jumps `<setjmp.h>` . . . 191

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

7.13.1 Save calling environment . . . 191
7.13.2 Restore calling environment . . . 191
7.14 Signal handling <signal.h> . . . 193
7.14.1 Specify signal handling . . . 193
7.14.2 Send signal . . . 194
7.15 Alignment <stdalign.h> . . . 196
7.16 Variable arguments <stdarg.h> . . . 197
7.16.1 Variable argument list access macros . . . 197
7.17 Atomics <stdatomic.h> . . . 200
7.17.1 Introduction . . . 200
7.17.2 Initialization . . . 201
7.17.3 Order and consistency . . . 201
7.17.4 Fences . . . 204
7.17.5 Lock-free property . . . 205
7.17.6 Atomic integer types . . . 205
7.17.7 Operations on atomic types . . . 206
7.17.8 Atomic flag type and operations . . . 208
7.18 Boolean type and values <stdbool.h> . . . 210
7.19 Common definitions <stddef.h> . . . 211
7.20 Integer types <stdint.h> . . . 212
7.20.1 Integer types . . . 212
7.20.2 Limits of specified-width integer types . . . 213
7.20.3 Limits of other integer types . . . 215
7.20.4 Macros for integer constants . . . 216
7.21 Input/output <stdio.h> . . . 217
7.21.1 Introduction . . . 217
7.21.2 Streams . . . 218
7.21.3 Files . . . 219
7.21.4 Operations on files . . . 221
7.21.5 File access functions . . . 222
7.21.6 Formatted input/output functions . . . 225
7.21.7 Character input/output functions . . . 241
7.21.8 Direct input/output functions . . . 244
7.21.9 File positioning functions . . . 245
7.21.10 Error-handling functions . . . 247
7.22 General utilities <stdlib.h> . . . 249
7.22.1 Numeric conversion functions . . . 249
7.22.2 Pseudo-random sequence generation functions . . . 253
7.22.3 Memory management functions . . . 254
7.22.4 Communication with the environment . . . 256

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

7.22.5 Searching and sorting utilities . . . 259
7.22.6 Integer arithmetic functions . . . 260
7.22.7 Multibyte/wide character conversion functions . . . 261
7.22.8 Multibyte/wide string conversion functions . . . 262
7.23 `_Noreturn` `<stdnoreturn.h>` . . . 264
7.24 String handling `<string.h>` . . . 265
7.24.1 String function conventions . . . 265
7.24.2 Copying functions . . . 265
7.24.3 Concatenation functions . . . 266
7.24.4 Comparison functions . . . 267
7.24.5 Search functions . . . 268
7.24.6 Miscellaneous functions . . . 271
7.25 Type-generic math `<tgmath.h>` . . . 273
7.26 Threads `<threads.h>` . . . 275
7.26.1 Introduction . . . 275
7.26.2 Initialization functions . . . 276
7.26.3 Condition variable functions . . . 276
7.26.4 Mutex functions . . . 278
7.26.5 Thread functions . . . 280
7.26.6 Thread-specific storage functions . . . 282
7.27 Date and time `<time.h>` . . . 285
7.27.1 Components of time . . . 285
7.27.2 Time manipulation functions . . . 286
7.27.3 Time conversion functions . . . 288
7.28 Unicode utilities `<uchar.h>` . . . 293
7.28.1 Restartable multibyte/wide character conversion functions . . . 293
7.29 Extended multibyte and wide character utilities `<wchar.h>` . . . 296
7.29.1 Introduction . . . 296
7.29.2 Formatted wide character input/output functions . . . 296
7.29.3 Wide character input/output functions . . . 308
7.29.4 General wide string utilities . . . 312
7.29.4.1 Wide string numeric conversion functions . . . 312
7.29.4.2 Wide string copying functions . . . 315
7.29.4.3 Wide string concatenation functions . . . 316
7.29.4.4 Wide string comparison functions . . . 316
7.29.4.5 Wide string search functions . . . 318
7.29.4.6 Miscellaneous functions . . . 321
7.29.5 Wide character time conversion functions . . . 321
7.29.6 Extended multibyte/wide character conversion utilities . . . 322
7.29.6.1 Single-byte/wide character conversion functions . . . 322

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

7.29.6.2 Conversion state functions 323
7.29.6.3 Restartable multibyte/wide character conversion functions 323
7.29.6.4 Restartable multibyte/wide string conversion functions 325
7.30 Wide character classification and mapping utilities <wctype.h> 327
7.30.1 Introduction 327
7.30.2 Wide character classification utilities 327
7.30.2.1 Wide character classification functions 327
7.30.2.2 Extensible wide character classification functions 330
7.30.3 Wide character case mapping utilities 331
7.30.3.1 Wide character case mapping functions 331
7.30.3.2 Extensible wide character case mapping functions 331
7.31 Future library directions 333
7.31.1 Complex arithmetic <complex.h> 333
7.31.2 Character handling <ctype.h> 333
7.31.3 Errors <errno.h> 333
7.31.4 Floating-point environment <fenv.h> 333
7.31.5 Format conversion of integer types <inttypes.h> 333
7.31.6 Localization <locale.h> 333
7.31.7 Signal handling <signal.h> 333
7.31.8 Atomics <stdatomic.h> 333
7.31.9 Boolean type and values <stdbool.h> 333
7.31.10 Integer types <stdint.h> 333
7.31.11 Input/output <stdio.h> 334
7.31.12 General utilities <stdlib.h> 334
7.31.13 String handling <string.h> 334
7.31.14 Date and time <time.h> 334
7.31.15 Threads <threads.h> 334
7.31.16 Extended multibyte and wide character utilities <wchar.h> 334
7.31.17 Wide character classification and mapping utilities <wctype.h> 334

Annex A (informative) Language syntax summary **335**

Annex B (informative) Library summary **347**

Annex C (informative) Sequence points **367**

Annex D (normative) Universal character names for identifiers **368**

Annex E (informative) Implementation limits **369**

Annex F (normative) IEC 60559 floating-point arithmetic **371**

Annex G (normative) IEC 60559-compatible complex arithmetic **389**

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

Annex H (informative) Language independent arithmetic 399

Annex I (informative) Common warnings 403

Annex J (informative) Portability issues 404

Annex K (normative) Bounds-checking interfaces 425

Annex L (normative) Analyzability 474

Annex M (informative) Change History 476

Bibliography 479

Index 480

Documents
Document Sharing Hub

Full document details: ISO IEC 09899-2018

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 The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

3 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. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

4 Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

5 For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see the following URL: www.iso.org/iso/foreword.html.

6 This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

7 This fourth edition cancels and replaces the third edition, ISO/IEC 9899:2011, which has been technically revised. It also incorporates the Technical Corrigendum ISO/IEC 9899:2011/Cor 1:2012.

8 There are no major changes in this edition, only technical corrections and clarifications.

9 A complete change history can be found in Annex M.

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

Introduction

1 With the introduction of new devices and extended character sets, new features could be added to this document. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, could conflict with future additions.

2 Certain features are *obsolescent*, which means that they could be considered for withdrawal in future revisions of this document. 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 document 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 document. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes define optional features, provide additional information and summarize the information contained in this document. A bibliography lists documents that were referred to during the preparation of this document.

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 document (WG 14) maintains a site on the World Wide Web at `http://www.open-std.org/JTC1/SC22/WG14/` containing ancillary information that may be of interest to some readers such as a Rationale for many of the decisions made during its preparation and a log of Defect Reports and Responses.

Documents
Document Sharing Hub

Full document details: ISO IEC 09899-2018

INTERNATIONAL STANDARD ©ISO/IEC ISO/IEC 9899:2018

Programming languages — C

1. Scope

1 This document 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 document 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 document 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. Annex J gives an overview of portability issues that a C program might encounter.

Documents Document Sharing Hub

Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

2. Normative references

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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 2382:2015, *Information technology — Vocabulary*. Available from the *ISO online browsing platform* at `http://www.iso.org/obp`

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)*. Available from the ISO/IEC Information Technology Task Force (ITTF) web site at `http://isotc.iso.org/livelink/livelink/fetch/2000/2489/Ittf_Home/PubliclyAvailableStandards.htm`

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*

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

3. Terms, definitions and symbols

1 For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO 80000–2, and the following apply.

2 ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at `https://www.iso.org/obp`

— IEC Electropedia: available at `http://www.electropedia.org/`

3 Additional terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this document are not to be presumed to refer implicitly to similar terms defined elsewhere.

3.1

1 **access (verb)**

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

2 **Note 1 to entry:** Where only one of these two actions is meant, "read" or "modify" is used.

3 **Note 2 to entry:** "Modify" includes the case where the new value being stored is the same as the previous value.

4 **Note 3 to entry:** 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

DEPRECATED: actual parameter

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 **Note 1 to entry:** J.3 gives an overview over properties of C programs that lead to implementation-defined behavior.

3 **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

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

2 **Note 1 to entry:** J.4 gives an overview over properties of C programs that lead to locale-specific behavior.

3 **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 document imposes no requirements

2 **Note 1 to entry:** 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 **Note 2 to entry:** J.2 gives an overview over properties of C programs that lead to undefined behavior.

4 **EXAMPLE** An example of undefined behavior is the behavior on integer overflow.

3.4.4

1 **unspecified behavior**

behavior, that results from the use of an unspecified value, or other behavior upon which this document provides two or more possibilities and imposes no further requirements on which is chosen in any instance

2 **Note 1 to entry:** J.1 gives an overview over properties of C programs that lead to unspecified behavior.

3 **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 can have one of two values

2 **Note 1 to entry:** 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 to entry:** It is possible to express the address of each individual byte of an object uniquely.

3 **Note 2 to entry:** 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 1 to entry:** The extended character set is a superset of the basic character set.

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

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

2 **Note 1 to entry:** In this document, when the words "correctly rounded" are not immediately followed by "result", this is the intended usage.

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 document 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 to entry:** Two threads of execution can update and access separate memory locations without interfering with each other.

3 **Note 2 to entry:** 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 (nonzero-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;
}
```


Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

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 1 to entry:** When referenced, an object can be interpreted as having a particular type; see 6.3.2.1.

3.16

1 **parameter**

formal parameter

DEPRECATED: formal argument

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 might be impractical for some implementations

3.18

1 **runtime-constraint**

requirement on a program when calling a library function

2 **Note 1 to entry:** 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 to entry:** 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.

4 **Note 3 to entry:** Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

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 document imposes no requirements on which value is chosen in any instance

2 **Note 1 to entry:** An unspecified value cannot be a trap representation.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

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 1 to entry:** In this document, when the word "trap" is not immediately followed by "representation", this is the intended usage.[2)]

3 **Note 2 to entry:** Implementations that support Annex L are permitted to invoke a runtime-constraint handler when they perform a trap.

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).

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

4. Conformance

1 In this document, "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 document 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 document.[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]

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).

[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 document.

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

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

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 document. 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 document. 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.

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 tokens[7)] 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.

[6)] This requires implementations to 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.

[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.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

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).

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 document describes the behavior for a construct as being both a constraint error and resulting in undefined behavior, the constraint error is still required to 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.

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

[9)] An implementation is encouraged to identify the nature of, and where possible localize, each violation. Of course, an implementation is free to produce any number of diagnostic messages, often referred to as warnings, as long as a valid program is still correctly translated. It can also successfully translate an invalid program. Annex I lists a few of the more common warnings.

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

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).

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 document 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

[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.

[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.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

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 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.

[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.

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

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 **int**s and truncate the sum. Provided the addition of two **char**s 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 can 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

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

```
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 document. 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 could 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.

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. Relaxed atomic operations are not included 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

[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.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

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 are expected to 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 can 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 dependency*[15)] to an evaluation B if:

— 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.

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

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

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

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. The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation.

19 **NOTE 8** This cycle would otherwise be possible only through the use of consume operations.

20 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.

21 **NOTE 9** 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.

22 **NOTE 10** 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.

23 The value of an atomic object M, as determined by evaluation B, shall be the value stored by some side effect A that modifies M, where B does not happen before A.

24 **NOTE 11** The set of side effects from which a given evaluation might take its value is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below.

25 If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M.

26 **NOTE 12** The requirement above is known as "write-write coherence".

27 If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M.

28 **NOTE 13** The requirement above is known as "read-read coherence".

29 If a value computation A of an atomic object M happens before an operation B on M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M.

30 **NOTE 14** The requirement above is known as "read-write coherence".

31 If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M.

32 **NOTE 15** The requirement above is known as "write-read coherence".

33 **NOTE 16** This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are "relaxed" loads. By doing so, it effectively makes the "cache coherence" guarantee provided by most hardware available to C atomic operations.

34 **NOTE 17** The value observed by a load of an atomic object depends on the "happens before" relation, which in turn depends on the values observed by loads of atomic objects. The intended reading is that there exists 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.

35 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

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

race results in undefined behavior.

36 **NOTE 18** 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 necessarily has undefined behavior even before such a transformation is applied.

37 **NOTE 19** 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 document, 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. Reordering of atomic loads in cases in which the atomics in question might alias is also generally precluded, since this could violate the coherence requirements.

38 **NOTE 20** Transformations that introduce a speculative read of a potentially shared memory location might not preserve the semantics of the program as defined in this document, 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.

Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the *Latin alphabet*

3
```
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 document 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

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

behavior is undefined.

4 A *letter* is an uppercase letter or a lowercase letter as defined above; in this document 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.

— 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.

[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.

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

— 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 document.

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

[18)] Implementations are encouraged to avoid imposing fixed translation limits whenever possible.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

— 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 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 **#include**d 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 (absolute value) to those shown, with the same sign.

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

[19)] See "future language directions" (6.11.3).

AVA Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

```
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)
```


Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

— 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 an object of type **char** can hold negative values, 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 $2^{\texttt{CHAR_BIT}} - 1$.

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] An implementation that defines **__STDC_IEC_559__** shall implement floating point types and arithmetic conforming to IEC 60559 as specified in Annex F. An implementation that defines **__STDC_IEC_559_COMPLEX__** shall implement complex types and arithmetic conforming to IEC 60559 as specified in Annex G.

2 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)

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

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

4 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 arithmetic operand.[22]

[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.

[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.

AVA Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

5 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 document to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored.

6 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.

7 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.

8 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**.

9 The rounding mode for floating-point addition is characterized by the implementation-defined value of **FLT_ROUNDS**:[23)]

−1	indeterminable
0	toward zero
1	to nearest
2	toward positive infinity
3	toward negative infinity

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

10 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	indeterminable;
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.

11 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**:

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

[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**.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

-1 indeterminable[25)]

0 absent (type does not support subnormal numbers)[26)]

1 present (type does support subnormal numbers)

12 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
```

— number of base-**FLT_RADIX** digits in the floating-point significand, p

```
FLT_MANT_DIG
DBL_MANT_DIG
LDBL_MANT_DIG
```

— number of decimal digits, n, such that any floating-point number with p radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

$$\begin{cases} p \log_{10} b & \text{if } b \text{ is a power of } 10 \\ \lceil 1 + p \log_{10} b \rceil & \text{otherwise} \end{cases}$$

```
FLT_DECIMAL_DIG                  6
DBL_DECIMAL_DIG                 10
LDBL_DECIMAL_DIG                10
```

— number of decimal digits, n, such that any floating-point number in the widest supported floating type with $p_{\max}$ radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value,

$$\begin{cases} p_{\max} \log_{10} b & \text{if } b \text{ is a power of } 10 \\ \lceil 1 + p_{\max} \log_{10} b \rceil & \text{otherwise} \end{cases}$$

```
DECIMAL_DIG                     10
```

— 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,

$$\begin{cases} p \log_{10} b & \text{if } b \text{ is a power of } 10 \\ \lfloor (p-1) \log_{10} b \rfloor & \text{otherwise} \end{cases}$$

```
FLT_DIG                          6
DBL_DIG                         10
LDBL_DIG                        10
```

[25)] Characterization as indeterminable 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.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

— minimum negative integer such that **FLT_RADIX** raised to one less than that power is a normalized floating-point number, $e_{\min}$

```
FLT_MIN_EXP
DBL_MIN_EXP
LDBL_MIN_EXP
```

— minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil log_{10} b^{e_{\min}-1} \rceil$

```
FLT_MIN_10_EXP                -37
DBL_MIN_10_EXP                -37
LDBL_MIN_10_EXP               -37
```

— maximum integer such that **FLT_RADIX** raised to one less than that power is a representable finite floating-point number, $e_{\max}$

```
FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP
```

— maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor log_{10}((1 - b^{-p}) b^{e_{\max}}) \rfloor$

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

13 The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater than or equal to those shown:

— maximum representable finite floating-point number, $(1 - b^{-p}) b^{e_{\max}}$

```
FLT_MAX                     1E+37
DBL_MAX                     1E+37
LDBL_MAX                    1E+37
```

14 The values given in the following list shall be replaced by constant expressions with implementation-defined (positive) values that are less than or equal to those shown:

— the difference between 1 and the least value greater than 1 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
```

— minimum positive floating-point number[27)]

[27)] If the presence or absence of subnormal numbers is indeterminable, then the value is intended to be a positive number no greater than the minimum normalized positive number for the type.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

```
FLT_TRUE_MIN                1E-37
DBL_TRUE_MIN                1E-37
LDBL_TRUE_MIN               1E-37
```

Recommended practice

15 Conversion from (at least) **double** to decimal with **DECIMAL_DIG** digits and back should be the identity function.

16 **EXAMPLE 1** The following describes an artificial floating-point representation that meets the minimum requirements of this document, and the appropriate values in a <float.h> header for type **float**:

$$x = s16^e \sum_{k=1}^{6} f_k 16^{-k}, \quad -31 \le e \le +32$$

```
FLT_RADIX                        16
FLT_MANT_DIG                      6
FLT_EPSILON          9.53674316E-07F
FLT_DECIMAL_DIG                   9
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
```

17 **EXAMPLE 2** The following describes floating-point representations that also meet the requirements for single-precision and double-precision numbers in IEC 60559,[28] and the appropriate values in a <float.h> header for types **float** and **double**:

$$x_f = s2^e \sum_{k=1}^{24} f_k 2^{-k}, \quad -125 \le e \le +128$$

$$x_d = s2^e \sum_{k=1}^{53} f_k 2^{-k}, \quad -1021 \le e \le +1024$$

```
FLT_RADIX                         2
DECIMAL_DIG                      17
FLT_MANT_DIG                     24
FLT_EPSILON          1.19209290E-07F // decimal constant
FLT_EPSILON                 0X1P-23F // hex constant
FLT_DECIMAL_DIG                   9
FLT_DIG                           6
FLT_MIN_EXP                    -125
FLT_MIN              1.17549435E-38F // decimal constant
FLT_MIN                    0X1P-126F // hex constant
FLT_TRUE_MIN         1.40129846E-45F // decimal constant
FLT_TRUE_MIN               0X1P-149F // hex constant
FLT_HAS_SUBNORM                   1
FLT_MIN_10_EXP                  -37
FLT_MAX_EXP                    +128
FLT_MAX              3.40282347E+38F // decimal constant
FLT_MAX              0X1.fffffeP127F // hex constant
FLT_MAX_10_EXP                  +38
DBL_MANT_DIG                     53
DBL_EPSILON   2.2204460492503131E-16 // decimal constant
DBL_EPSILON                 0X1P-52 // hex constant
DBL_DECIMAL_DIG                  17
DBL_DIG                          15
```

[28] 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.

AVA Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

```
DBL_MIN_EXP                    -1021
DBL_MIN      2.2250738585072014E-308 // decimal constant
DBL_MIN                   0X1P-1022 // hex constant
DBL_TRUE_MIN 4.9406564584124654E-324 // decimal constant
DBL_TRUE_MIN              0X1P-1074 // hex constant
DBL_HAS_SUBNORM                   1
DBL_MIN_10_EXP                 -307
DBL_MAX_EXP                    +1024
DBL_MAX      1.7976931348623157E+308 // decimal constant
DBL_MAX       0X1.fffffffffffffP1023 // hex constant
DBL_MAX_10_EXP                 +308
```

If a type wider than **double** were supported, then **DECIMAL_DIG** would be greater than 17. For example, if the widest type were to use the minimal-width IEC 60559 double-extended format (64 bits of precision), then **DECIMAL_DIG** would be 21.

Forward references: conditional inclusion (6.10.1), predefined macro names (6.10.8), complex arithmetic `<complex.h>` (7.3), extended multibyte and wide character utilities `<wchar.h>` (7.29), floating-point environment `<fenv.h>` (7.6), general utilities `<stdlib.h>` (7.22), input/output `<stdio.h>` (7.21), mathematics `<math.h>` (7.12), IEC 60559 floating-point arithmetic (Annex F), IEC 60559-compatible complex arithmetic (Annex G).

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

6. Language

6.1 Notation

1 In the syntax notation used in this clause, 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. 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

$$\texttt{\{}\ \textit{expression}_{\text{opt}}\ \texttt{\}}$$

indicates an optional expression enclosed in braces.

2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

3 A summary of the language syntax is given in Annex A.

6.2 Concepts

6.2.1 Scopes of identifiers

1 An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. The same identifier can denote different entities at different points in the program. A member of an enumeration is called an *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.

2 For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. Different entities designated by the same identifier either have different scopes, or are in different name spaces. 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.)

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

4 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 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 end of 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 terminates at the end of the function declarator. If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the *inner scope*) will end strictly before the scope of the other entity (the *outer scope*). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

5 Unless explicitly stated otherwise, where this document uses the term "identifier" to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

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

7 Structure, union, and enumeration tags have scope that begins just after the appearance of the 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

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

begins just after the completion of its declarator.

8 As a special case, a type name (which is not a declaration of an identifier) is considered to have a scope that begins just after the place within the type name where the omitted identifier would appear were it not omitted.

Forward references: declarations (6.7), function calls (6.5.2.2), function definitions (6.9.1), identifiers (6.4.2), macro replacement (6.10.3), name spaces of identifiers (6.2.3), source file inclusion (6.10.2), statements and blocks (6.8).

6.2.2 Linkages of identifiers

1 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*.[29] There are three kinds of linkage: external, internal, and none.

2 In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. Within one translation unit, each declaration of an identifier with *internal linkage* denotes the same object or function. Each declaration of an identifier with *no linkage* denotes a unique entity.

3 If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **`static`**, the identifier has internal linkage.[30]

4 For an identifier declared with the storage-class specifier **`extern`** in a scope in which a prior declaration of that identifier is visible,[31] if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

5 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.

6 The following identifiers have no linkage: an identifier declared to be anything other than an 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`**.

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

Forward references: declarations (6.7), expressions (6.5), external definitions (6.9), statements (6.8).

6.2.3 Name spaces of identifiers

1 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:

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

— the *tags* of structures, unions, and enumerations (disambiguated by following any[32] 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 **`.`** or **`->`** operator);

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

[29] There is no linkage between different identifiers.
[30] A function declaration can contain the storage-class specifier **`static`** only if it is at file scope; see 6.7.1.
[31] As specified in 6.2.1, the later declaration might hide the prior declaration.
[32] There is only one name space for tags even though three are possible.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

Forward references: enumeration specifiers (6.7.2.2), labeled statements (6.8.1), structure and union specifiers (6.7.2.1), structure and union members (6.5.2.3), tags (6.7.2.3), the`goto` statement (6.8.6.1).

6.2.4 Storage durations of objects

1 An object has a *storage duration* that determines its lifetime. There are four storage durations: static, thread, automatic, and allocated. Allocated storage is described in 7.22.3.

2 The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address,[33] and retains its last-stored value throughout its lifetime.[34] If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to (or just past) reaches the end of its lifetime.

3 An object whose identifier is declared without the storage-class specifier `_Thread_local`, and either with external or internal linkage or with the storage-class specifier `static`, has *static storage duration*. Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

4 An object whose identifier is declared with the storage-class specifier `_Thread_local` has *thread storage duration*. Its lifetime is the entire execution of the thread for which it is created, and its stored value is initialized when the thread is started. There is a distinct object per thread, and use of the declared name in an expression refers to the object associated with the thread evaluating the expression. The result of attempting to indirectly access an object with thread storage duration from a thread other than the one with which the object is associated is implementation-defined.

5 An object whose identifier is declared with no linkage and without the storage-class specifier `static` has *automatic storage duration*, as do some compound literals. The result of attempting to indirectly access an object with automatic storage duration from a thread other than the one with which the object is associated is implementation-defined.

6 For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) If the block is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration or compound literal is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached.

7 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[35] If the scope is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate.

8 A non-lvalue expression with structure or union type, where the structure or union contains a member with array type (including, recursively, members of all contained structures and unions) refers to an object with automatic storage duration and *temporary lifetime*.[36] Its lifetime begins when the expression is evaluated and its initial value is the value of the expression. Its lifetime ends when the evaluation of the containing full expression ends. Any attempt to modify an object with temporary lifetime results in undefined behavior. An object with temporary lifetime behaves as if it were declared with the type of its value for the purposes of effective type. Such an object need not have a unique address.

Forward references: array declarators (6.7.6.2), compound literals (6.5.2.5), declarators (6.7.6), function calls (6.5.2.2), initialization (6.7.9), statements (6.8), effective type (6.5).

[33] The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal. The address can be different during two different executions of the same program.

[34] In the case of a volatile object, the last store need not be explicit in the program.

[35] Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration.

[36] The address of such an object is taken implicitly when an array member is accessed.

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

6.2.5 Types

1 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) and *function types* (types that describe functions). At various points within a translation unit an object type may be *incomplete* (lacking sufficient information to determine the size of objects of that type) or *complete* (having sufficient information).[37)]

2 An object declared as type **_Bool** is large enough to store the values 0 and 1.

3 An object declared as type **char** is large enough to store any member of the basic execution character set. If a member of the basic execution character set is stored in a **char** object, its value is guaranteed to be nonnegative. If any other character is stored in a **char** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type.

4 There are five *standard signed integer types*, designated as **signed char**, **short int**, **int**, **long int**, and **long long int**. (These and other types may be designated in several additional ways, as described in 6.7.2.) There may also be implementation-defined *extended signed integer types*.[38)] The standard and extended signed integer types are collectively called *signed integer types*.[39)]

5 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>`).

6 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 type **_Bool** and the unsigned integer types that correspond to the standard signed integer types are the *standard unsigned integer types*. The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. The standard and extended unsigned integer types are collectively called *unsigned integer types*.[40)]

7 The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*; the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

8 For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type.

9 The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same. [41)] 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 type.

10 There are three *real floating types*, designated as **float**, **double**, and **long double**.[42)] The set 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**.

11 There are three *complex types*, designated as **float _Complex**, **double _Complex**, and **long double _Complex**.[43)] (Complex types are a conditional feature that implementations need not support; see

[37)] A type can be incomplete or complete throughout an entire translation unit, or it can change states at different points within a translation unit.

[38)] Implementation-defined keywords have the form of an identifier reserved for any use as described in 7.1.3.

[39)] Therefore, any statement in this document about signed integer types also applies to the extended signed integer types.

[40)] Therefore, any statement in this document about unsigned integer types also applies to the extended unsigned integer types.

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

[42)] See "future language directions" (6.11.1).

[43)] A specification for imaginary types is in Annex G.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

6.10.8.3.) The real floating and complex types are collectively called the *floating types*.

12 For each floating type there is a *corresponding real type*, which is always a real floating type. For real floating types, it is the same type. For complex types, it is the type given by deleting the keyword **_Complex** from the type name.

13 Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.

14 The type **char**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. The basic types are complete object types. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.[44)]

15 The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*. The implementation shall define **char** to have the same range, representation, and behavior as either **signed char** or **unsigned char**.[45)]

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

17 The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integer types*. The integer and real floating types are collectively called *real types*.

18 Integer and floating types are collectively called *arithmetic types*.[46)] Each arithmetic type belongs to one *type domain*: the *real type domain* comprises the real types, the *complex type domain* comprises the complex types.

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

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

— An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. The element type shall be complete whenever the array type is specified. Array types are characterized by their element 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 (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.

— 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 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 or an object 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 *T* is sometimes called "pointer to *T*".

[44)] An implementation can define new keywords that provide alternative ways to designate a basic (or any other) type; this does not violate the requirement that all basic types be different. Implementation-defined keywords have the form of an identifier reserved for any use as described in 7.1.3.

[45)] **CHAR_MIN**, defined in <limits.h>, will have one of the values 0 or **SCHAR_MIN**, and this can be used to distinguish the two options. Irrespective of the choice made, **char** is a separate type from the other two and is not compatible with either.

[46)] Annex H documents the extent to which the C language supports the ISO/IEC 10967–1 standard for language-independent arithmetic (LIA–1).

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

The construction of a pointer type from a referenced type is called "pointer type derivation". A pointer type is a complete object type.

— An *atomic type* describes the type designated by the construct **_Atomic**(*type-name*). (Atomic types are a conditional feature that implementations need not support; see 6.10.8.3.)

These methods of constructing derived types can be applied recursively.

21 Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.[47)]

22 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 or union type of unknown content (as described in 6.7.2.3) 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.

23 A type has *known constant size* if the type is not incomplete and is not a variable length array type.

24 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* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

25 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.

26 Any type so far mentioned is an *unqualified type*. Each unqualified type has several *qualified versions* of its type,[48)] corresponding to the combinations of one, two, or all three of the **const**, **volatile**, and **restrict** qualifiers. 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.[49)] A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

27 Further, there is the **_Atomic** qualifier. The presence of the **_Atomic** qualifier designates an atomic type. The size, representation, and alignment of an atomic type need not be the same as those of the corresponding unqualified type. Therefore, this document explicitly uses the phrase "atomic, qualified or unqualified type" whenever the atomic version of a type is permitted along with the other qualified versions of a type. The phrase "qualified or unqualified type", without specific mention of atomic, does not include the atomic types.

28 A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type.[49)] Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same representation or alignment requirements.

29 **EXAMPLE 1** 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.

30 **EXAMPLE 2** The type designated as "**struct tag (*[5])(float)**" has type "array of 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: compatible type and composite type (6.2.7), declarations (6.7).

6.2.6 Representations of types

6.2.6.1 General

1 The representations of all types are unspecified except as stated in this subclause.

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

[48)] See 6.7.3 regarding qualified array and function types.

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

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

2 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.

3 Values stored in unsigned bit-fields and objects of type **unsigned char** shall be represented using a pure binary notation.[50)]

4 Values stored in non-bit-field objects of any other object type consist of $n \times$ **CHAR_BIT** bits, where n is the size of an object of that type, in bytes. The value may be copied into an object of type **unsigned char [**n**]** (e.g., by **memcpy**); the resulting set of bytes is called the *object representation* of the value. Values stored in bit-fields consist of m bits, where m is the size specified for the bit-field. The object representation is the set of m bits the bit-field comprises in the addressable storage unit holding it. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.

5 Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[51)] Such a representation is called a trap representation.

6 When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values. [52)] The value of a structure or union object is never a trap representation, even though the value of a member of the structure or union object may be a trap representation.

7 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.

8 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.[53)] Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated.

9 Loads and stores of objects with atomic types are done with **memory_order_seq_cst** semantics.

Forward references: declarations (6.7), expressions (6.5), lvalues, arrays, and function designators (6.3.2.1), order and consistency (7.17.3).

6.2.6.2 Integer types

1 For unsigned integer types other than **unsigned char**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to $2^N - 1$ using a pure binary representation; this shall be known as the value representation. The values of any padding bits are unspecified.[54)]

2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. There need not be any padding bits; **signed char** shall not have any padding bits. There shall be exactly one sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if

[50)] 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*.) A byte contains **CHAR_BIT** bits, and the values of type **unsigned char** range from 0 to $2^{\texttt{CHAR_BIT}} - 1$.

[51)] Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it.

[52)] Thus, for example, structure assignment need not copy any padding bits.

[53)] It is possible for objects **x** and **y** with the same effective type **T** to have the same value when they are accessed as objects of type **T**, but to have different values in other contexts. In particular, if **==** is defined for type **T**, then **x == y** does not imply that **memcmp(&x, &y, sizeof (T))== 0**. Furthermore, **x == y** does not necessarily imply that **x** and **y** have the same value; other operations on values of type **T** might distinguish between them.

[54)] Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

there are M value bits in the signed type and N in the unsigned type, then $M \leq N$). If the sign bit is zero, it shall not affect the resulting value. If the sign bit is one, the value shall be modified in one of the following ways:

— the corresponding value with sign bit 0 is negated (*sign and magnitude*);

— the sign bit has the value $-(2^M)$ (*two's complement*);

— the sign bit has the value $-(2^M - 1)$ (*ones' complement*).

Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for ones' complement), is a trap representation or a normal value. In the case of sign and magnitude and ones' complement, if this representation is a normal value it is called a *negative zero*.

3 If the implementation supports negative zeros, they shall be generated only by:

— the **&**, **|**, **^**, **~**, **<<**, and **>>** operators with operands that produce such a value;

— the **+**, **-** , *****, **/**, and **%** operators where one operand is a negative zero and the result is zero;

— compound assignment operators based on the above cases.

It is unspecified whether these cases actually generate a negative zero or a normal zero, and whether a negative zero becomes a normal zero when stored in an object.

4 If the implementation does not support negative zeros, the behavior of the **&**, **|**, **^**, **~**, **<<**, and **>>** operators with operands that would produce such a value is undefined.

5 The values of any padding bits are unspecified.[55] A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.

6 The *precision* of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. The *width* of an integer type is the same but including any sign bit; thus for unsigned integer types the two values are the same, while for signed integer types the width is one greater than the precision.

6.2.7 Compatible type and composite type

1 Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.6 for declarators.[56] Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: If one is declared with a tag, the other shall be declared with the same tag. If both are completed anywhere within their respective translation units, then the following additional requirements apply: there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types; if one member of the pair is declared with an alignment specifier, the other is declared with an equivalent alignment specifier; and if one member of the pair is declared with a name, the other is declared with the same name. For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding members shall have the same values.

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

[55] Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow. All other combinations of padding bits are alternative object representations of the value specified by the value bits.

[56] Two types need not be identical to be compatible.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

3 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 both types are array types, the following rules are applied:

- If one type is an array of known constant size, the composite type is an array of that size.
- Otherwise, if one type is a variable length array whose size is specified by an expression that is not evaluated, the behavior is undefined.
- Otherwise, if one type is a variable length array whose size is specified, the composite type is a variable length array of that size.
- Otherwise, if one type is a variable length array of unspecified size, the composite type is a variable length array of unspecified size.
- Otherwise, both types are arrays of unknown size and the composite type is an array of unknown size.

The element type of the composite type is the composite type of the two element types.

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

4 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible,[57] if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type.

Forward references: array declarators (6.7.6.2).

5 **EXAMPLE** Given the following two file scope declarations:

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

The resulting composite type for the function is:

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

6.2.8 Alignment of objects

1 Complete object types have alignment requirements which place restrictions on the addresses at which objects of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type: stricter alignment can be requested using the **_Alignas** keyword.

2 A *fundamental alignment* is a valid alignment less than or equal to **_Alignof (max_align_t)**. Fundamental alignments shall be supported by the implementation for objects of all storage durations. The alignment requirements of the following types shall be fundamental alignments:

— all atomic, qualified or unqualified basic types;

— all atomic, qualified or unqualified enumerated types;

— all atomic, qualified or unqualified pointer types;

— all array types whose element type has a fundamental alignment requirement;

[57] As specified in 6.2.1, the later declaration might hide the prior declaration.

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018

— all types specified in Clause 7 as complete object types;

— all structure or union types all of whose elements have types with fundamental alignment requirements and none of whose elements have an alignment specifier specifying an alignment that is not a fundamental alignment.

3 An *extended alignment* is represented by an alignment greater than `_Alignof (max_align_t)`. It is implementation-defined whether any extended alignments are supported and the storage durations for which they are supported. A type having an extended alignment requirement is an *over-aligned* type.[58)]

4 Alignments are represented as values of the type `size_t`. Valid alignments include only fundamental alignments, plus an additional implementation-defined set of values, which may be empty. Every valid alignment value shall be a nonnegative integral power of two.

5 Alignments have an order from *weaker* to *stronger* or *stricter* alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

6 The alignment requirement of a complete type can be queried using an `_Alignof` expression. The types `char`, `signed char`, and `unsigned char` shall have the weakest alignment requirement.

7 Comparing alignments is meaningful and provides the obvious results:

— Two alignments are equal when their numeric values are equal.

— Two alignments are different when their numeric values are not equal.

— When an alignment is larger than another it represents a stricter alignment.

6.3 Conversions

1 Several operators convert operand values from one type to another automatically. This subclause 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 6.3.1.8 summarizes the conversions performed by most ordinary operators; it is supplemented as required by the discussion of each operator in 6.5.

2 Unless explicitly stated otherwise, conversion of an operand value to a compatible type causes no change to the value or the representation.

Forward references: cast operators (6.5.4).

6.3.1 Arithmetic operands

6.3.1.1 Boolean, characters, and integers

1 Every integer type has an *integer conversion rank* defined as follows:

— No two signed integer types shall have the same rank, even if they have the same representation.

— The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.

— The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.

— The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.

— The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.

[58)] Every over-aligned type is, or contains, a structure or union type with a member to which an extended alignment has been applied.

Documents Document Sharing Hub
Full document details: ISO IEC 09899-2018

 © ISO/IEC 2018– All rights reserved

— The rank of **char** shall equal the rank of **signed char** and **unsigned char**.

— The rank of **_Bool** shall be less than the rank of all other standard integer types.

— The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).

— The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.

— For all integer types **T1**, **T2**, and **T3**, if **T1** has greater rank than **T2** and **T2** has greater rank than **T3**, then **T1** has greater rank than **T3**.

2 The following may be used in an expression wherever an **int** or **unsigned int** may be used:

— An object or expression with an integer type (other than **int** or **unsigned int**) whose integer conversion rank is less than or equal to the rank of **int** and **unsigned int**.

— A bit-field of type **_Bool**, **int**, **signed int**, or **unsigned int**.

If an **int** can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an **int**; otherwise, it is converted to an **unsigned int**. These are called the *integer promotions*.[59] All other types are unchanged by the integer promotions.

3 The integer promotions preserve value including sign. As discussed earlier, whether a "plain" **char** can hold negative values is implementation-defined.

Forward references: enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1).

6.3.1.2 Boolean type

1 When any scalar value is converted to **_Bool**, the result is 0 if the value compares equal to 0; otherwise, the result is 1.[60]

6.3.1.3 Signed and unsigned integers

1 When a value with integer type is converted to another integer type other than **_Bool**, if the value can be represented by the new type, it is unchanged.

2 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.[61]

3 Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

6.3.1.4 Real floating and integer

1 When a finite value of real floating type is converted to an integer type other than **_Bool**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the behavior is undefined.[62]

2 When a value of integer type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. 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 representable value, chosen in an implementation-defined manner.

[59] The integer 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 subclauses.

[60] NaNs do not compare equal to 0 and thus convert to 1.

[61] The rules describe arithmetic on the mathematical value, not the value of a given type of expression.

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

Documents
Document Sharing Hub
Full document details: ISO IEC 09899-2018