International Standard

ISO/IEC 9899

Fifth edition
2024-10

Information technology — Programming languages — C

Technologies de l'information — Langages de programmation — C

Reference number
ISO/IEC 9899:2024(en)

© ISO/IEC 2024

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2024

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
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

Contents

Foreword **xii**

Introduction **xiii**

1 Scope **1**

2 Normative references **2**

3 Terms, definitions, and symbols **3**

4 Conformance **9**

5 Environment **11**
5.1 Introduction . . . 11
5.2 Conceptual models . . . 11
5.2.1 Translation environment . . . 11
5.2.2 Execution environments . . . 12
5.3 Environmental considerations . . . 19
5.3.1 Character sets . . . 19
5.3.2 Multibyte characters . . . 20
5.3.3 Character display semantics . . . 21
5.3.4 Signals and interrupts . . . 21
5.3.5 Environmental limits . . . 21

6 Language **35**
6.1 Notation . . . 35
6.2 Concepts . . . 35
6.2.1 Scopes of identifiers, type names, and compound literals . . . 35
6.2.2 Linkages of identifiers . . . 36
6.2.3 Name spaces of identifiers . . . 37
6.2.4 Storage durations of objects . . . 37
6.2.5 Types . . . 38
6.2.6 Representations of types . . . 42
6.2.7 Compatible type and composite type . . . 43
6.2.8 Alignment of objects . . . 45
6.2.9 Encodings . . . 45
6.3 Conversions . . . 46
6.3.1 Introduction . . . 46
6.3.2 Arithmetic operands . . . 46

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

6.3.3 Other operands . . . 49
6.4 Lexical elements . . . 52
6.4.1 General . . . 52
6.4.2 Keywords . . . 53
6.4.3 Identifiers . . . 54
6.4.4 Universal character names . . . 56
6.4.5 Constants . . . 57
6.4.6 String literals . . . 67
6.4.7 Punctuators . . . 68
6.4.8 Header names . . . 69
6.4.9 Preprocessing numbers . . . 70
6.4.10 Comments . . . 70
6.5 Expressions . . . 72
6.5.1 General . . . 72
6.5.2 Primary expressions . . . 73
6.5.3 Postfix operators . . . 74
6.5.4 Unary operators . . . 81
6.5.5 Cast operators . . . 83
6.5.6 Multiplicative operators . . . 84
6.5.7 Additive operators . . . 85
6.5.8 Bitwise shift operators . . . 86
6.5.9 Relational operators . . . 86
6.5.10 Equality operators . . . 87
6.5.11 Bitwise AND operator . . . 88
6.5.12 Bitwise exclusive OR operator . . . 88
6.5.13 Bitwise inclusive OR operator . . . 89
6.5.14 Logical AND operator . . . 89
6.5.15 Logical OR operator . . . 89
6.5.16 Conditional operator . . . 90
6.5.17 Assignment operators . . . 91
6.5.18 Comma operator . . . 94
6.6 Constant expressions . . . 95
6.7 Declarations . . . 97
6.7.1 General . . . 97
6.7.2 Storage-class specifiers . . . 98
6.7.3 Type specifiers . . . 103
6.7.4 Type qualifiers . . . 120
6.7.5 Function specifiers . . . 124
6.7.6 Alignment specifier . . . 125
6.7.7 Declarators . . . 126

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

6.7.8 Type names . . . 132
6.7.9 Type definitions . . . 133
6.7.10 Type inference . . . 134
6.7.11 Initialization . . . 136
6.7.12 Static assertions . . . 142
6.7.13 Attributes . . . 142
6.8 Statements and blocks . . . 152
6.8.1 General . . . 152
6.8.2 Labeled statements . . . 153
6.8.3 Compound statement . . . 153
6.8.4 Expression and null statements . . . 153
6.8.5 Selection statements . . . 154
6.8.6 Iteration statements . . . 155
6.8.7 Jump statements . . . 156
6.9 External definitions . . . 159
6.9.1 General . . . 159
6.9.2 Function definitions . . . 159
6.9.3 External object definitions . . . 161
6.10 Preprocessing directives . . . 163
6.10.1 General . . . 163
6.10.2 Conditional inclusion . . . 165
6.10.3 Source file inclusion . . . 169
6.10.4 Binary resource inclusion . . . 171
6.10.5 Macro replacement . . . 178
6.10.6 Line control . . . 185
6.10.7 Diagnostic directives . . . 186
6.10.8 Pragma directive . . . 186
6.10.9 Null directive . . . 187
6.10.10 Predefined macro names . . . 187
6.10.11 Pragma operator . . . 189
6.11 Future language directions . . . 190
6.11.1 Floating types . . . 190
6.11.2 Linkages of identifiers . . . 190
6.11.3 External names . . . 190
6.11.4 Character escape sequences . . . 190
6.11.5 Storage-class specifiers . . . 190
6.11.6 Pragma directives . . . 190
6.11.7 Predefined macro names . . . 190

7 Library **191**

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

7.1 Introduction . . . 191
7.1.1 Definitions of terms . . . 191
7.1.2 Standard headers . . . 191
7.1.3 Reserved identifiers . . . 192
7.1.4 Use of library functions . . . 193
7.2 Diagnostics `<assert.h>` . . . 195
7.2.1 General . . . 195
7.2.2 Program diagnostics . . . 195
7.3 Complex arithmetic `<complex.h>` . . . 196
7.3.1 Introduction . . . 196
7.3.2 Conventions . . . 196
7.3.3 Branch cuts . . . 197
7.3.4 The **CX_LIMITED_RANGE** pragma . . . 197
7.3.5 Trigonometric functions . . . 197
7.3.6 Hyperbolic functions . . . 199
7.3.7 Exponential and logarithmic functions . . . 200
7.3.8 Power and absolute-value functions . . . 201
7.3.9 Manipulation functions . . . 202
7.4 Character handling `<ctype.h>` . . . 205
7.4.1 General . . . 205
7.4.2 Character classification functions . . . 205
7.4.3 Character case mapping functions . . . 207
7.5 Errors `<errno.h>` . . . 209
7.6 Floating-point environment `<fenv.h>` . . . 210
7.6.1 General . . . 210
7.6.2 The **FENV_ACCESS** pragma . . . 212
7.6.3 The **FENV_ROUND** pragma . . . 213
7.6.4 The **FENV_DEC_ROUND** pragma . . . 215
7.6.5 Floating-point exceptions . . . 216
7.6.6 Rounding and other control modes . . . 218
7.6.7 Environment . . . 221
7.7 Characteristics of floating types `<float.h>` . . . 223
7.8 Format conversion of integer types `<inttypes.h>` . . . 224
7.8.1 General . . . 224
7.8.2 Macros for format specifiers . . . 224
7.8.3 Functions for greatest-width integer types . . . 225
7.9 Alternative spellings `<iso646.h>` . . . 227
7.10 Characteristics of integer types `<limits.h>` . . . 228
7.11 Localization `<locale.h>` . . . 229
7.11.1 General . . . 229

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

7.11.2 The **setlocale** function . . . 229
7.11.3 Numeric formatting convention inquiry . . . 230
7.12 Mathematics <math.h> . . . 235
7.12.1 General . . . 235
7.12.2 Treatment of error conditions . . . 238
7.12.3 The **FP_CONTRACT** pragma . . . 239
7.12.4 Classification macros . . . 239
7.12.5 Trigonometric functions . . . 242
7.12.6 Hyperbolic functions . . . 247
7.12.7 Exponential and logarithmic functions . . . 249
7.12.8 Power and absolute-value functions . . . 257
7.12.9 Error and gamma functions . . . 260
7.12.10 Nearest integer functions . . . 262
7.12.11 Remainder functions . . . 266
7.12.12 Manipulation functions . . . 268
7.12.13 Maximum, minimum, and positive difference functions . . . 270
7.12.14 Fused multiply-add . . . 275
7.12.15 Functions that round result to narrower type . . . 275
7.12.16 Quantum and quantum exponent functions . . . 277
7.12.17 Decimal re-encoding functions . . . 279
7.12.18 Comparison macros . . . 281
7.13 Non-local jumps <setjmp.h> . . . 284
7.13.1 General . . . 284
7.13.2 Save calling environment . . . 284
7.13.3 Restore calling environment . . . 284
7.14 Signal handling <signal.h> . . . 286
7.14.1 General . . . 286
7.14.2 Specify signal handling . . . 286
7.14.3 Send signal . . . 288
7.15 Alignment <stdalign.h> . . . 289
7.16 Variable arguments <stdarg.h> . . . 290
7.16.1 General . . . 290
7.16.2 Variable argument list access macros . . . 290
7.17 Atomics <stdatomic.h> . . . 294
7.17.1 Introduction . . . 294
7.17.2 Initialization . . . 295
7.17.3 Order and consistency . . . 296
7.17.4 Fences . . . 298
7.17.5 Lock-free property . . . 299
7.17.6 Atomic integer types . . . 300

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

7.17.7 Operations on atomic types . . . 301
7.17.8 Atomic flag type and operations . . . 303
7.18 Bit and byte utilities `<stdbit.h>` . . . 305
7.18.1 General . . . 305
7.18.2 Endian . . . 305
7.18.3 Count Leading Zeros . . . 306
7.18.4 Count Leading Ones . . . 306
7.18.5 Count Trailing Zeros . . . 306
7.18.6 Count Trailing Ones . . . 307
7.18.7 First Leading Zero . . . 307
7.18.8 First Leading One . . . 308
7.18.9 First Trailing Zero . . . 308
7.18.10 First Trailing One . . . 309
7.18.11 Count Zeros . . . 310
7.18.12 Count Ones . . . 310
7.18.13 Single-bit Check . . . 310
7.18.14 Bit Width . . . 311
7.18.15 Bit Floor . . . 311
7.18.16 Bit Ceiling . . . 312
7.19 Boolean type and values `<stdbool.h>` . . . 313
7.20 Checked Integer Arithmetic `<stdckdint.h>` . . . 314
7.20.1 General . . . 314
7.20.2 Checked Integer Operation Type-generic Macros . . . 314
7.21 Common definitions `<stddef.h>` . . . 315
7.21.1 General . . . 315
7.21.2 The **`unreachable`** macro . . . 316
7.21.3 The **`nullptr_t`** type . . . 317
7.22 Integer types `<stdint.h>` . . . 318
7.22.1 General . . . 318
7.22.2 Integer types . . . 318
7.22.3 Widths of specified-width integer types . . . 320
7.22.4 Width of other integer types . . . 320
7.22.5 Macros for integer constants . . . 321
7.22.6 Maximal and minimal values of integer types . . . 321
7.23 Input/output `<stdio.h>` . . . 322
7.23.1 Introduction . . . 322
7.23.2 Streams . . . 324
7.23.3 Files . . . 325
7.23.4 Operations on files . . . 326
7.23.5 File access functions . . . 328

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

7.23.6 Formatted input/output functions . . . 331
7.23.7 Character input/output functions . . . 349
7.23.8 Direct input/output functions . . . 352
7.23.9 File positioning functions . . . 353
7.23.10 Error-handling functions . . . 355
7.24 General utilities `<stdlib.h>` . . . 357
7.24.1 General . . . 357
7.24.2 Numeric conversion functions . . . 357
7.24.3 Pseudo-random sequence generation functions . . . 364
7.24.4 Memory management functions . . . 365
7.24.5 Communication with the environment . . . 368
7.24.6 Searching and sorting utilities . . . 371
7.24.7 Integer arithmetic functions . . . 372
7.24.8 Multibyte/wide character conversion functions . . . 373
7.24.9 Multibyte/wide string conversion functions . . . 374
7.24.10 Alignment of memory . . . 375
7.25 **`_Noreturn`** `<stdnoreturn.h>` . . . 377
7.26 String handling `<string.h>` . . . 378
7.26.1 String function conventions . . . 378
7.26.2 Copying functions . . . 378
7.26.3 Concatenation functions . . . 380
7.26.4 Comparison functions . . . 381
7.26.5 Search functions . . . 382
7.26.6 Miscellaneous functions . . . 385
7.27 Type-generic math `<tgmath.h>` . . . 387
7.28 Threads `<threads.h>` . . . 392
7.28.1 Introduction . . . 392
7.28.2 Initialization functions . . . 393
7.28.3 Condition variable functions . . . 393
7.28.4 Mutex functions . . . 395
7.28.5 Thread functions . . . 397
7.28.6 Thread-specific storage functions . . . 399
7.29 Date and time `<time.h>` . . . 402
7.29.1 Components of time . . . 402
7.29.2 Time manipulation functions . . . 403
7.29.3 Time conversion functions . . . 406
7.30 Unicode utilities `<uchar.h>` . . . 412
7.30.1 General . . . 412
7.30.2 Restartable multibyte/wide character conversion functions . . . 412
7.31 Extended multibyte and wide character utilities `<wchar.h>` . . . 417

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

7.31.1 Introduction . . . 417
7.31.2 Formatted wide character input/output functions . . . 418
7.31.3 Wide character input/output functions . . . 431
7.31.4 General wide string utilities . . . 435
7.31.4.1 General . . . 435
7.31.4.2 Wide string numeric conversion functions . . . 435
7.31.4.3 Wide string copying functions . . . 440
7.31.4.4 Wide string concatenation functions . . . 441
7.31.4.5 Wide string comparison functions . . . 441
7.31.4.6 Wide string search functions . . . 443
7.31.4.7 Miscellaneous functions . . . 446
7.31.5 Wide character time conversion functions . . . 446
7.31.6 Extended multibyte/wide character conversion utilities . . . 447
7.31.6.1 General . . . 447
7.31.6.2 Single-byte/wide character conversion functions . . . 447
7.31.6.3 Conversion state functions . . . 448
7.31.6.4 Restartable multibyte/wide character conversion functions . . . 448
7.31.6.5 Restartable multibyte/wide string conversion functions . . . 450
7.32 Wide character classification and mapping utilities `<wctype.h>` . . . 452
7.32.1 Introduction . . . 452
7.32.2 Wide character classification utilities . . . 452
7.32.2.1 General . . . 452
7.32.2.2 Wide character classification functions . . . 452
7.32.2.3 Extensible wide character classification functions . . . 455
7.32.3 Wide character case mapping utilities . . . 456
7.32.3.1 Wide character case mapping functions . . . 456
7.32.3.2 Extensible wide character case mapping functions . . . 456
7.33 Future library directions . . . 458
7.33.1 General . . . 458
7.33.2 Complex arithmetic `<complex.h>` . . . 458
7.33.3 Character handling `<ctype.h>` . . . 458
7.33.4 Errors `<errno.h>` . . . 458
7.33.5 Floating-point environment `<fenv.h>` . . . 458
7.33.6 Characteristics of floating types `<float.h>` . . . 458
7.33.7 Format conversion of integer types `<inttypes.h>` . . . 458
7.33.8 Localization `<locale.h>` . . . 458
7.33.9 Mathematics `<math.h>` . . . 458
7.33.10 Signal handling `<signal.h>` . . . 459
7.33.11 Atomics `<stdatomic.h>` . . . 459
7.33.12 Boolean type and values `<stdbool.h>` . . . 459

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

7.33.13 Bit and byte utilities <stdbit.h> 459
7.33.14 Checked Arithmetic Functions <stdckdint.h> 459
7.33.15 Integer types <stdint.h> 459
7.33.16 Input/output <stdio.h> 459
7.33.17 General utilities <stdlib.h> 459
7.33.18 String handling <string.h> 460
7.33.19 Date and time <time.h> 460
7.33.20 Threads <threads.h> 460
7.33.21 Extended multibyte and wide character utilities <wchar.h> 460
7.33.22 Wide character classification and mapping utilities <wctype.h> 460

Annex A (informative) Language syntax summary **461**

Annex B (informative) Library summary **476**

Annex C (informative) Sequence points **516**

Annex D (informative) Universal character names for identifiers **517**

Annex E (informative) Implementation limits **519**

Annex F (normative) ISO/IEC 60559 floating-point arithmetic **522**

Annex G (normative) ISO/IEC 60559-compatible complex arithmetic **553**

Annex H (normative) ISO/IEC 60559 interchange and extended types **564**

Annex I (informative) Common warnings **597**

Annex J (informative) Portability issues **598**

Annex K (normative) Bounds-checking interfaces **636**

Annex L (normative) Analyzability **684**

Annex M (informative) Change History **685**

Bibliography **691**

Index **692**

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members 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.

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 or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

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

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 www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC1, Information technology, Subcommittee SC22, Programming languages, their environments and system software interfaces.

This fifth edition cancels and replaces the fourth edition (ISO/IEC 9899:2018), which has been technically revised. The main changes are contained in Annex M.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

Introduction

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

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 [7.33]) is discouraged.

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).

In any given subsequent clause or subclause, there are section delineations in **bold** to describe the semantics, restrictions, and behaviors of programs for this language and potentially the use of its library clauses in this document:

— **Syntax**
which pertains to the spelling and organization of the language and library;

— **Constraints**
which detail and enumerate various requirements for the correct interpretation of the language and library, typically during translation;

— **Semantics**
which explain the behavior of language features and similar constructs;

— **Description**
which explain the behavior of library usage and similar constructs;

— **Returns**
which describes the effects of constructs provided back to a user of the library;

— **Runtime-constraints**
which detail and enumerate various requirements that are expected to be checked and which shall not be violated, typically during execution;

— **Environmental limits**
which list limitations an implementation may impose on a library or language construct which might otherwise be unlimited;

— **Recommended practice**
which provides guidance and important considerations for implementers of this document.

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 implementers. Annexes define optional features, provide additional

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

information and summarize the information contained in this document. A bibliography lists documents that were referred to during the preparation of this document.

The language clause (Clause 6) is derived from "The C Reference Manual"[15].

The library clause (Clause 7) is based on the *1984 /usr/group Standard*[16].

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

INTERNATIONAL STANDARD ©ISO/IEC ISO/IEC 9899:2024

Information technology — Programming languages — C

1. Scope

This document specifies the form and establishes the interpretation of programs written in the C programming language. It is designed to promote the portability of C programs among a variety of data-processing systems. It is intended for use by implementers and programmers. 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.

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.

Annex J gives an overview of portability issues that a C program can encounter.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

2. Normative references

The following documents are referred to in the text in such a way that some or all 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.

ISO/IEC 2382:2015, *Information technology — Vocabulary*.

ISO 4217, *Codes for the representation of currencies*.

ISO 8601 series, *Data elements and interchange formats — Information interchange — Representation of dates and times*.

ISO/IEC 10646, *Information technology —Universal Coded Character Set (UCS)*.

ISO/IEC 60559:2020, *Information technology — Microprocessor Systems — Floating-Point arithmetic*.

ISO 80000–2, *Quantities and units — Part 2: Mathematics*.

The Unicode Consortium. *Unicode Standard Annex, UAX #44, Unicode Character Database [online]*. Edited by Ken Whistler. Available at `https://www.unicode.org/reports/tr44`.

The Unicode Consortium. *The Unicode Standard, Derived Core Properties*. Available at `https://www.unicode.org/Public/UCD/latest/ucd/DerivedCoreProperties.txt`.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

3. Terms, definitions, and symbols

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

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

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

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

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
access (verb)

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

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

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

Note 3 to entry: Expressions that are not evaluated do not access objects.

3.2
alignment

requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte (3.7) address

3.3
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
arithmetically negate

produce the negative of a given number

Note 1 to entry: For a floating-point number (5.3.5.3.3), this changes the sign; for an integer, this is equivalent to subtracting from zero.

3.5
behavior

external appearance or action

3.5.1
implementation-defined behavior

unspecified behavior where each implementation documents how the choice is made

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

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

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

3.5.2
locale-specific behavior

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

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

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

3.5.3
undefined behavior

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

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).

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

Note 3 to entry: Any other behavior during execution of a program is only affected as a direct consequence of the concrete behavior that occurs when encountering the erroneous or non-portable program construct or data. In particular, all observable behavior (5.2.2.4) appears as specified in this document when it happens before an operation with undefined behavior in the execution of the program.

EXAMPLE An example of undefined behavior is the behavior on dereferencing a null pointer.

3.5.4
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

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

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

3.6
bit

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

Note 1 to entry: It is possible that each individual bit of an object is not adressable.

3.7
byte

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

Note 1 to entry: Each individual byte of an object is uniquely addressable.

3.8
low-order bit

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

the least significant bit within a byte

3.9
high-order bit

the most significant bit within a byte

3.10
character

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

3.10.1
character

single-byte character

⟨C⟩ bit representation that fits in a byte

3.10.2
multibyte character

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

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

3.10.3
wide character

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

3.11
constraint

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

3.12
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

Note 1 to entry: In this document, the words "correctly rounded" may apply to an operation that produces a correctly rounded result, or to input for such an operation.

Note 2 to entry: ISO/IEC 60559 or implementation-defined rules apply for extreme magnitude results if the result format contains infinity.

3.13
diagnostic message

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

3.14
forward reference

reference to a subsequent subclause in this document that contains additional information relevant to the subclause containing the reference

3.15
implementation

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

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

3.16
implementation limit

restriction imposed upon programs by the implementation

3.17
memory location

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

Note 1 to entry: Two threads of execution can update and access separate memory locations without interfering with each other.

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.

EXAMPLE A structure declared as

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

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

3.18
object

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

Note 1 to entry: When referenced, an object can be interpreted as having a particular type; see 6.3.3.1.

3.19
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

3.20
recommended practice

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

3.21
runtime-constraint

requirement on a program when calling a library function

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

Note 1 to entry: Despite the similar terms, a runtime-constraint is not a kind of constraint as defined by 3.11, and it is not necessary for it to be diagnosed at translation time.

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.

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

3.22
value

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

3.22.1
implementation-defined value

unspecified value where each implementation documents how the choice is made

3.22.2
unspecified value

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

3.23
indeterminate representation

object representation that either represents an unspecified value or is a non-value representation

3.24
non-value representation

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

3.25
perform a trap

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

Note 1 to entry: Fetching a non-value representation permits an implementation to perform a trap but is not required to (see 6.2.6.1).

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

3.26
$\lceil x \rceil$

ceiling of x

the least integer greater than or equal to x

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

3.27
$\lfloor x \rfloor$

floor of x

the greatest integer less than or equal to x

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

3.28
wraparound

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

the process by which a value is reduced modulo 2^N, where N is the width of the resulting type

3.29

out-of-bounds store

(attempted) access (3.1) that, at run time, for a given computational state, would modify (or, for an object declared **`volatile`**, fetch) one or more bytes that lie outside the bounds permitted by this document.

3.30

bounded undefined behavior

undefined behavior (3.5.3) that does not perform an out-of-bounds store.

Note 1 to entry: The behavior can perform a trap.

Note 2 to entry: Any values produced can be unspecified values, and the representation of objects that are written to can become indeterminate.

3.31

critical undefined behavior

undefined behavior that is not bounded undefined behavior.

Note 1 to entry: The behavior can perform an out-of-bounds store or perform a trap.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

4. Conformance

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.

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".

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 5.2.2.4.

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.

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

EXAMPLE A strictly conforming program can use conditional features (see 6.10.10.4) provided the use is guarded by an appropriate conditional inclusion preprocessing directive using the related macro. For example:

```
#ifdef __STDC_IEC_60559_BFP__ /* FE_UPWARD defined */
        /* ... */
        fesetround(FE_UPWARD);
        /* ... */
#endif
```

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>`, `<stdbit.h>`, `<stdbool.h>`, `<stddef.h>`, `<stdint.h>`, and `<stdnoreturn.h>`. Additionally, a *conforming freestanding implementation* shall accept any strictly conforming program where:

— the features specified in the header `<string.h>` are used, except the following functions: **`strcoll`**, **`strdup`**, **`strerror`**, **`strndup`**, **`strtok`**, **`strxfrm`**; and/or,

— the selected function **`memalignment`** from `<stdlib.h>` is used.

A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.[1)]

The strictly conforming programs that shall be accepted by a conforming freestanding implementation that defines **`__STDC_IEC_60559_BFP__`** or **`__STDC_IEC_60559_DFP__`** can also use features in the contents of the standard headers `<fenv.h>`, `<math.h>`, and the **`strto`**∗ floating-point numeric conversion functions (7.24.2) of the standard header `<stdlib.h>`, provided the program does not set the state of the **`FENV_ACCESS`** pragma to "on".

All identifiers that are reserved when `<stdlib.h>` is included in a hosted implementation are reserved when it is included in a freestanding implementation.

A *conforming program* is one that is acceptable to a conforming implementation.[2)]

[1)]This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this document.

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

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

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

Forward references: conditional inclusion (6.10.2), error directive (6.10.7), 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>` (7.19), common definitions `<stddef.h>` (7.21), integer types `<stdint.h>` (7.22), `<stdnoreturn.h>` (7.25).

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

5. Environment

5.1 Introduction

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.

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

5.2 Conceptual models

5.2.1 Translation environment

5.2.1.1 Program structure

A C program is not required to be translated in its entirety 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 can 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).

5.2.1.2 Translation phases

The precedence among the syntax rules of translation is specified by the following phases.[3)]

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.

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[4)] 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 (6.10.5.4), 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.

[3)] 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 necessarily can be stored as files or through/within any other implementation-defined medium. There is no expectation of any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation.

[4)] 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.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

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. Each instance of a source character or escape sequence for which there is no corresponding member is converted in an implementation-defined manner to some member of the execution character set other than the null (wide) character.[5)]
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 (6.4.4), lexical elements (6.4), preprocessing directives (6.10), external definitions (6.9).

5.2.1.3 Diagnostics

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 are not required to be produced in other circumstances.

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.

Recommended practice

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.

5.2.2 Execution environments

5.2.2.1 General

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 (6.7.11).

5.2.2.2 Freestanding environment

In a freestanding environment (in which C program execution can 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.

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

[5)] An implementation can convert each instance of the same non-corresponding source character to a different member of the execution character set.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

5.2.2.3 Hosted environment

5.2.2.3.1 General

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

5.2.2.3.2 Program startup

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 can be used, as they are local to the function in which they are declared):

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

or equivalent;[6] or in some other implementation-defined manner.

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.

5.2.2.3.3 Program execution

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

5.2.2.3.4 Program termination

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;[7] 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 (7.24.5.4).

[6] Thus, **int** can be replaced by a typedef name defined as **int**, or the type of **argv** can be written as **char ** argv**, or the return type can be specified by **typeof(1)**, and so on.

[7] 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.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

5.2.2.4 Program semantics

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

An access to an object through the use of an lvalue of volatile-qualified type is a *volatile access*. A volatile access to an object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*,[8)] 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.

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.[9)] 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.)

In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation is not required to 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 through volatile access to an object).

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 dynamic floating-point environment. The representation 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 dynamic floating-point environment if it is modified by the handler and not restored to its original state.

The least requirements on a conforming implementation are:

— Volatile accesses to 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 7.23.3. The intent of these requirements is that unbuffered or line-buffered output appear as soon as possible, to ensure that prompting messages appear prior to a program waiting for input.

This is the *observable behavior* of the program.

What constitutes an interactive device is implementation-defined.

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

EXAMPLE 1 An implementation can 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.

Alternatively, an implementation can 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

[8)] ISO/IEC 60559 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.

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

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

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.

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

EXAMPLE 3 Similarly, in the fragment

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

the multiplication can be executed using **float** arithmetic if the implementation can ascertain that the result would be the same as if it were executed using **double** arithmetic (for example, if **d** were replaced by the constant **2.0**, which has type **double**).

EXAMPLE 4 Implementations employing wide registers are expected 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 can not alter the value. Also, an explicit *store and load* rounds to the precision the precision of the storage type. In particular, casts and assignments 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**.

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 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;
```

EXAMPLE 6 To illustrate the grouping behavior of expressions, in the following fragment

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

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

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 integer overflows produce an explicit trap and in which the range of values representable by an **int** is $[-32768, +32767]$, the implementation cannot rewrite this expression as

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

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

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

or

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

since the values for **a** and **b** can be, respectively, 4 and -8 or -17 and 12. However, on an implementation in which integer overflow silently generates some value and where positive and negative integer overflows cancel, the preceding expression statement can be rewritten by the implementation in any of the previously specified ways because the same result will occur.

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 (6.5.1), type qualifiers (6.7.4), statements (6.8), floating-point environment `<fenv.h>` (7.6), the **signal** function (7.14), files (7.23.3).

5.2.2.5 Multi-threaded executions and data races

Under a hosted implementation that does not define **__STDC_NO_THREADS__**, 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 its threads.[10] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

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 in the rest of this subclause.

NOTE 1 In some cases, there can 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.

Two expression evaluations *conflict* if one of them modifies a memory location and the other one reads or modifies the same memory location.

[10] The execution can usually be viewed as an interleaving of all the threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving as described in this subclause.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

The library defines *atomic operations* (7.17) and operations on mutexes (7.28.4) 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 one of 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.

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.

All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M. If A and B are modifications of an atomic object M, and A happens before B, then A shall precede B in the modification order of M, which is defined later in this subclause.

NOTE 3 This states that the modification orders are expected to respect the "happens before" relation.

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.

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.

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.

NOTE 5 Except in the specified cases, reading a later value does not necessarily ensure visibility as described later in this subclause. Such a requirement would sometimes interfere with efficient implementation.

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.

An evaluation A *carries a dependency*[11)] 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.

An evaluation A is *dependency-ordered before*[12)] an evaluation B if:

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

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

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

— 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.

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.

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 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 subsequently in this subclause, provides for relationships consisting entirely of "sequenced before".

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.

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

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.

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.

NOTE 10 This states that operations on ordinary variables are not visibly reordered. This is not detectable without data races, but ensures 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.

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.

NOTE 11 The set of side effects from which a given evaluation can take its value is also restricted by the rest of the rules described here, and in particular, by the coherence requirements subsequently in this subclause.

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.

NOTE 12 Such a requirement is known as "write-write coherence".

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.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

NOTE 13 Such a requirement is known as "read-read coherence".

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.

NOTE 14 Such a requirement is known as "read-write coherence".

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.

NOTE 15 Such a requirement is known as "write-read coherence".

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.

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 previously, satisfy the resulting constraints as imposed here.

The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.

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 of such transformations necessarily has undefined behavior even before such a transformation is applied.

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 can 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 can alias is also generally precluded, since this can violate the coherence requirements.

NOTE 20 Transformations that introduce a speculative read of a potentially shared memory location possibly will 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.

5.3 Environmental considerations

5.3.1 Character sets

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.

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*

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

```
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 *digit*

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

the following 32 *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 preceding 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 behavior is undefined.

A *letter* is an uppercase letter or a lowercase letter as defined previously in this subclause; in this document the term does not include other characters that are letters in other alphabets.

The universal character name construct provides a way to name other characters.

Forward references: universal character names (6.4.4), character constants (6.4.5.5), preprocessing directives (6.10), string literals (6.4.6), comments (6.4.10), string (7.1.1).

5.3.2 Multibyte characters

The source character set can contain multibyte characters, used to represent members of the extended character set. The execution character set can also contain multibyte characters, which are not required to 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 can 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.

For source files, the following shall hold:

— An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state.

— An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

5.3.3 Character display semantics

The *active position* is that location on a display device where the next character output by the **fputc** function would appear. The intent of writing a 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.

Alphabetic escape sequences representing non-graphic 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.

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 are not necessarily identical to the internal representations, and are outside the scope of this document.

Forward references: the **isprint** function (7.4.2.9), the **fputc** function (7.23.7.3).

5.3.4 Signals and interrupts

Functions shall be implemented such that they can be interrupted at any time by a signal, or can 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.

5.3.5 Environmental limits

5.3.5.1 General

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.

5.3.5.2 Translation limits

The implementation shall be able to translate and execute a program that uses but does not exceed the following limitations for these constructs and entities:[13)]

— 127 nesting levels of blocks

— 63 nesting levels of conditional inclusion

— 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or **void** type in a declaration

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

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

— 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 00FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 010000 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)[14)]

— 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)

— 32767 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 member declaration list

5.3.5.3 Numerical limits

5.3.5.3.1 General

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>` (7.22).

5.3.5.3.2 Characteristics of integer types `<limits.h>`

The values given subsequently shall be replaced by constant expressions suitable for use in conditional expression inclusion preprocessing directives. Their implementation-defined values shall be equal or greater to those shown.

— width for an object of type **`bool`**[15)]

```
BOOL_WIDTH                      1
```

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

[14)] See "future language directions" (6.11.3).
[15)] This value is exact.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

```
CHAR_BIT                        8
```

The macros **CHAR_WIDTH**, **SCHAR_WIDTH**, and **UCHAR_WIDTH** that represent the width of the types **char**, **signed char** and **unsigned char** shall expand to the same value as **CHAR_BIT**.

— width for an object of type **unsigned short int**

```
USHRT_WIDTH                    16
```

The macro **SHRT_WIDTH** represents the width of the type **short int** and shall expand to the same value as **USHRT_WIDTH**.

— width for an object of type **unsigned int**

```
UINT_WIDTH                     16
```

The macro **INT_WIDTH** represents the width of the type **int** and shall expand to the same value as **UINT_WIDTH**.

— width for an object of type **unsigned long int**

```
ULONG_WIDTH                     32
```

The macro **LONG_WIDTH** represents the width of the type **long int** and shall expand to the same value as **ULONG_WIDTH**.

— width for an object of type **unsigned long long int**

```
ULLONG_WIDTH                    64
```

The macro **LLONG_WIDTH** represents the width of the type **long long int** and shall expand to the same value as **ULLONG_WIDTH**.

— maximum width of a bit-precise integer type

```
BITINT_MAXWIDTH     /* see the following */
```

The macro **BITINT_MAXWIDTH** represents the maximum width N supported by the declaration of a bit-precise integer (6.2.5) in the type specifier **_BitInt(**N**)**. The value **BITINT_MAXWIDTH** shall expand to a value that is greater than or equal to the value of **ULLONG_WIDTH**.

— maximum number of bytes in a multibyte character, for any supported locale

```
MB_LEN_MAX                      1
```

For all unsigned integer types for which <limits.h> or <stdint.h> define a macro with suffix **_WIDTH** holding its width N, there is a macro with suffix **_MAX** holding the maximal value $2^N - 1$ that is representable by the type and that has the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. If the value is in the range of the type **uintmax_t** (7.22.2.6) the macro is suitable for use in conditional expression inclusion preprocessing directives.

For all signed integer types for which <limits.h> or <stdint.h> define a macro with suffix **_WIDTH** holding its width N, there are macros with suffix **_MIN** and **_MAX** holding the minimal and maximal values -2^{N-1} and $2^{N-1} - 1$ that are representable by the type and that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. If the values are in the range of the type **intmax_t** (7.22.2.6) the macros are suitable for use in conditional expression inclusion preprocessing directives.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

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** (see 6.2.5.).

Forward references: representations of types (6.2.6), conditional inclusion (6.10.2), integer types `<stdint.h>` (7.22).

5.3.5.3.3 Characteristics of floating types `<float.h>`

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and allows other values. The characteristics provide information about an implementation's floating-point arithmetic.[16] An implementation that defines **__STDC_IEC_60559_BFP__** or **__STDC_IEC_559__** shall implement floating types and arithmetic conforming to ISO/IEC 60559 as specified in Annex F of this document. An implementation that defines **__STDC_IEC_60559_COMPLEX__** or **__STDC_IEC_559_COMPLEX__** shall implement complex types and arithmetic conforming to ISO/IEC 60559 as specified in Annex G of this document.

The following parameters are used to define the model for each floating 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)

For each floating type, the parameters b, p, $e_{\min}$, and $e_{\max}$ are fixed constants.

For each floating type, a *floating-point number* (x) is defined by the following model:

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

Model floating-point numbers x with $f_1 > 0$ are called *normalized floating-point numbers*.

Model floating-point numbers $x \neq 0$ with $f_1 = 0$ and $e = e_{\min}$ are called *subnormal floating-point numbers*.

Model floating-point numbers $x \neq 0$ with $f_1 = 0$ and $e > e_{\min}$ are called *unnormalized floating-point numbers*.

Model floating-point numbers x with all $f_k = 0$ are zeros.

Floating types shall be able to represent signed zeros or an unsigned zero and all normalized floating-point numbers. In addition, floating types may be able to contain other kinds of floating-point numbers, such as subnormal floating-point numbers and unnormalized floating-point numbers, and values that are not floating-point numbers, such as NaNs and (signed and unsigned) infinities.

NOTE 1 Some implementations have types that include finite numbers with range and/or precision that are not covered by the model.

A *NaN* is a value 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.

NOTE 2 ISO/IEC 60559 specifies quiet and signaling NaNs. For implementations that do not support ISO/IEC 60559, the terms quiet NaN and signaling NaN are intended to apply to values with similar behavior.

Wherever values are unsigned, any requirement in this document to get the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored, unless otherwise specified.

NOTE 3 Bit representations of floating-point values can include a sign bit, even if the values can be regarded as unsigned; ISO/IEC 60559 NaNs are such values.

Whether and in what cases subnormal numbers are treated as zeros is implementation-defined. Subnormal numbers that in some cases are treated by arithmetic operations as zeros are properly

[16] 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.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

classified as subnormal. However, object representations that could represent subnormal numbers but that are always treated by arithmetic operations as zeros are non-canonical zeros, and the values are properly classified as zero, not subnormal. ISO/IEC 60559 arithmetic (with default exception handling) always treats subnormal numbers as nonzero.

A value is negative if and only if it compares less than 0. Thus, negative zeros and NaNs are not negative values.

An implementation can prefer particular representations of values that have multiple representations in a floating type, 6.2.6.1 not withstanding. The preferred representations of a floating type, including unique representations of values in the type, are called *canonical*. A floating type can also contain *non-canonical* representations, for example, redundant representations of some or all its values, or representations that are extraneous to the floating-point model. Typically, floating-point operations deliver results with canonical representations. ISO/IEC 60559 operations deliver results with canonical representations, unless specified otherwise.

NOTE 4 The library operations **`iscanonical`** and **`canonicalize`** distinguish canonical (preferred) representations, but this distinction alone does not imply that canonical and non-canonical representations are of different values.

NOTE 5 Some of the values in the ISO/IEC 60559 decimal formats have non-canonical representations (as well as a canonical representation).

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.

The accuracy of the floating-point operations (`+`, `-`, `*`, `/`) and of most 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. Decimal floating-point operations have stricter requirements.

All integer values in the `<float.h>` header, except **`FLT_ROUNDS`**, shall be constant expressions suitable for use in conditional expression inclusion preprocessing directives; all floating values shall be arithmetic constant expressions. All except **`CR_DECIMAL_DIG`** (F.5), **`DECIMAL_DIG`**, **`DEC_EVAL_METHOD`**, **`FLT_EVAL_METHOD`**, **`FLT_RADIX`**, and **`FLT_ROUNDS`** have separate names for all floating types. The floating-point model representation is provided for all values except **`DEC_EVAL_METHOD`**, **`FLT_EVAL_METHOD`** and **`FLT_ROUNDS`**.

The remainder of this subclause specifies characteristics of standard floating types.

The rounding mode for floating-point addition for standard floating types is characterized by the implementation-defined value of **`FLT_ROUNDS`**. Evaluation of **`FLT_ROUNDS`** correctly reflects any execution-time change of rounding mode through the function **`fesetround`** in `<fenv.h>`.

−1	indeterminable
0	toward zero
1	to nearest, ties to even
2	toward positive infinity
3	toward negative infinity
4	to nearest, ties away from zero

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

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

Whether a type has the same base (b), precision (p), and exponent range ($e_{\min} \leq e \leq e_{\max}$) as an ISO/IEC 60559 format is characterized by the implementation-defined values of **FLT_IS_IEC_60559**, **DBL_IS_IEC_60559** and **LDBL_IS_IEC_60559** (this does not imply conformance to Annex F):

0 type does not have the precision and exponent range of an ISO/IEC 60559 format

1 type has the precision and exponent range of an ISO/IEC 60559 format

NOTE 6 Outside of the normalized floating-point numbers, the representability of values (e.g. negative zero) of the ISO/IEC 60559 format is not implied.

The values of floating type yielded by operators subject to the usual arithmetic conversions, including the values yielded by the implicit conversion of operands, and the values of floating constants are evaluated to a format whose range and precision can be greater than required by the type. Such a format is called an *evaluation format*. In all cases, assignment and cast operators yield values in the format of the type. The extent to which evaluation formats are used is characterized by the value of **FLT_EVAL_METHOD**:[17)]

−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. The value of **FLT_EVAL_METHOD** does not characterize values returned by function calls (see 6.8.7.5, F.6).

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

−1 indeterminable

0 absent (type does not support subnormal numbers)

1 present (type does support subnormal numbers)

The use of **FLT_HAS_SUBNORM**, **DBL_HAS_SUBNORM**, and **LDBL_HAS_SUBNORM** macros is an obsolescent feature.

Each of the signaling NaN macros

```
FLT_SNAN
DBL_SNAN
LDBL_SNAN
```

is defined if and only if the respective type contains signaling NaNs. They expand to a constant expression of the respective type representing a signaling NaN. If an optional unary **+** or **-** operator followed by a signaling NaN macro is used as an initializer that is evaluated at translation time, the object is initialized with a signaling NaN value.

The macro

```
INFINITY
```

[17)] 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**.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

is defined if and only if the implementation supports an infinity for the type **float**. It expands to a constant expression of type **float** representing positive or unsigned infinity.

The macro

```
NAN
```

is defined if and only if the implementation supports quiet NaNs for the **float** type. It expands to a constant expression of type **float** representing a quiet NaN.

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 of the supported floating types and the supported ISO/IEC 60559 encodings 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
```

This is an obsolescent feature, see 7.33.9.

— 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
```

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

— 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; if that representable finite floating-point number is normalized, the value of the macro is $e_{\max}$

```
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
```

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; if that number is normalized, its value is $(1 - b^{-p})b^{e_{\max}}$

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

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

```
FLT_NORM_MAX                 1E+37
DBL_NORM_MAX                 1E+37
LDBL_NORM_MAX                1E+37
```

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 normalized value greater than 1 that is representable in the given floating type, b^{1-p}

```
FLT_EPSILON                   1E-5
DBL_EPSILON                   1E-9
LDBL_EPSILON                  1E-9
```

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

— 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

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

Recommended practice

Conversion between real floating type and decimal character sequence with at most *T*`_DECIMAL_DIG` digits should be correctly rounded, where *T* is the macro prefix for the type. This assures conversion from real floating type to decimal character sequence with *T*`_DECIMAL_DIG` digits and back, using to-nearest rounding, is the identity function.

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
```

EXAMPLE 2 The following describes floating-point representations that also meet the requirements for binary32 and binary64 numbers in ISO/IEC 60559,[18] and the appropriate values in a <float.h> header for types `float` and `double`. The decimal floating constants can possibly not give correct values (and hence are not appropriate values in a `<float.h>` header) if `FLT_EVAL_METHOD` is not 0 or if a translation-time rounding mode other than the ISO/IEC 60559 default is supported (either as the default or as a constant rounding mode set by an `FENV_ROUND` pragma). The hexadecimal floating constants are correct in all such cases because their values are exactly representable in the type.

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

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

```
FLT_IS_IEC_60559                  1
FLT_RADIX                         2
FLT_MANT_DIG                     24
FLT_EPSILON          1.19209290E-07F // decimal constant
FLT_EPSILON                0X1P-23F // hex constant
FLT_DECIMAL_DIG                   9
FLT_DIG                           6
```

[18] The floating-point model in ISO/IEC 60559 sums powers of *b* from zero, so the values of the exponent limits are one less than shown here.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

```
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_IS_IEC_60559                   1
DBL_EPSILON   2.2204460492503131E-16 // decimal constant
DBL_EPSILON                  0X1P-52 // hex constant
DBL_DECIMAL_DIG                   17
DBL_DIG                           15
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
```

Forward references: conditional inclusion (6.10.2), predefined macro names (6.10.10), complex arithmetic `<complex.h>` (7.3), extended multibyte and wide character utilities `<wchar.h>` (7.31), floating-point environment `<fenv.h>` (7.6), general utilities `<stdlib.h>` (7.24), input/output `<stdio.h>` (7.23), mathematics `<math.h>` (7.12), ISO/IEC 60559 floating-point arithmetic (Annex F), ISO/IEC 60559-compatible complex arithmetic (Annex G).

5.3.5.3.4 Characteristics of decimal floating types in `<float.h>`

This subclause specifies macros in `<float.h>` that provide characteristics of decimal floating types (an optional feature) in terms of the model presented in 5.3.5.3.3. An implementation shall provide these macros if and only if it defines **`__STDC_IEC_60559_DFP__`**. The prefixes **`DEC32_`**, **`DEC64_`**, and **`DEC128_`** denote the types **`_Decimal32`**, **`_Decimal64`**, and **`_Decimal128`** respectively.

`DEC_EVAL_METHOD` is the decimal floating-point analog of **`FLT_EVAL_METHOD`** (5.3.5.3.3). Its implementation-defined value characterizes the use of evaluation formats for decimal floating types:

−1 indeterminable;

0 evaluate all operations and constants just to the range and precision of the type;

1 evaluate operations and constants of type **`_Decimal32`** and **`_Decimal64`** to the range and precision of the **`_Decimal64`** type, evaluate **`_Decimal128`** operations and constants to the range and precision of the **`_Decimal128`** type;

2 evaluate all operations and constants to the range and precision of the **`_Decimal128`** type.

Each of the decimal signaling NaN macros

```
DEC32_SNAN
DEC64_SNAN
DEC128_SNAN
```

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

expands to a constant expression of the respective decimal floating type representing a signaling NaN. If an optional unary **+** or **-** operator followed by a signaling NaN macro is used for initializing an object of the same type that has static or thread storage duration, the object is initialized with a signaling NaN value.

The macro

```
DEC_INFINITY
```

expands to a constant expression of type **_Decimal32** representing positive infinity.

The macro

```
DEC_NAN
```

expands to a constant expression of type **_Decimal32** representing a quiet NaN.

The integer values given in the following lists shall be replaced by constant expressions suitable for use in conditional expression inclusion preprocessing directives:

— radix of exponent representation, b(=10)

For the standard floating types, this value is implementation-defined and is specified by the macro **FLT_RADIX**. For the decimal floating types there is no corresponding macro, since the value 10 is an inherent property of the types. Wherever **FLT_RADIX** appears in a description of a function that has versions that operate on decimal floating types, it is noted that for the decimal floating-point versions the value used is implicitly 10, rather than **FLT_RADIX**.

— number of digits in the coefficient

```
DEC32_MANT_DIG     7
DEC64_MANT_DIG     16
DEC128_MANT_DIG    34
```

— minimum exponent

```
DEC32_MIN_EXP      -94
DEC64_MIN_EXP      -382
DEC128_MIN_EXP     -6142
```

— maximum exponent

```
DEC32_MAX_EXP      97
DEC64_MAX_EXP      385
DEC128_MAX_EXP     6145
```

— maximum representable finite decimal floating-point number (there are 6, 15 and 33 **9**'s after the decimal points respectively)

```
DEC32_MAX          9.999999E96DF
DEC64_MAX          9.999999999999999E384DD
DEC128_MAX         9.999999999999999999999999999999999E6144DL
```

— the difference between 1 and the least value greater than 1 that is representable in the given floating type

```
DEC32_EPSILON      1E-6DF
DEC64_EPSILON      1E-15DD
DEC128_EPSILON     1E-33DL
```

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

— minimum normalized positive decimal floating-point number

```
DEC32_MIN          1E-95DF
DEC64_MIN          1E-383DD
DEC128_MIN         1E-6143DL
```

— minimum positive subnormal decimal floating-point number

```
DEC32_TRUE_MIN     0.000001E-95DF
DEC64_TRUE_MIN     0.000000000000001E-383DD
DEC128_TRUE_MIN    0.0000000000000000000000000000000001E-6143DL
```

For decimal floating-point arithmetic, it is often convenient to consider an alternate equivalent model where the significand is represented with integer rather than fraction digits. With s, b, e, p, and f_k as defined in 5.3.5.3.3, a floating-point number x is defined by the model:

$$x = s \cdot b^{(e-p)} \sum_{k=1}^{p} f_k \cdot b^{(p-k)}$$

With b fixed to 10, a decimal floating-point number x is thus:

$$x = s \cdot 10^{(e-p)} \sum_{k=1}^{p} f_k \cdot 10^{(p-k)}$$

The *quantum exponent* is $q = e - p$ and the *coefficient* is $c = f_1 f_2 \cdots f_p$, which is an integer between 0 and $10^p - 1$, inclusive. Thus, $x = s \cdot c \cdot 10^q$ is represented by the triple of integers (s, c, q). The *quantum* of x is 10^q, which is the value of a unit in the last place of the coefficient. Table 5.1 shows the range of quantum exponents.

Table 5.1 — Quantum exponent ranges

Type	`_Decimal32`	`_Decimal64`	`_Decimal128`
Maximum Quantum Exponent ($q_{\max}$)	90	369	6111
Minimum Quantum Exponent ($q_{\min}$)	-101	-398	-6176

For binary floating-point arithmetic following ISO/IEC 60559, representations in the model described in 5.3.5.3.3 that have the same numerical value are indistinguishable in the arithmetic. However, for decimal floating-point arithmetic, representations that have the same numerical value but different quantum exponents, e.g. $(+1, 10, -1)$ representing 1.0 and $(+1, 100, -2)$ representing 1.00, are distinguishable. To facilitate exact fixed-point calculation, operation results that are of decimal floating type have a *preferred quantum exponent*, as specified in ISO/IEC 60559, which is determined by the quantum exponents of the operands if they have decimal floating types (or by specific rules for conversions from other types). Table 5.2 gives rules for determining preferred quantum exponents for results of ISO/IEC 60559 operations, and for other operations specified in this document. When exact, these operations produce a result with their preferred quantum exponent, or as close to it as possible within the limitations of the type. When inexact, these operations produce a result with the least possible quantum exponent. For example, the preferred quantum exponent for addition is the minimum of the quantum exponents of the operands. Hence $(+1, 123, -2) + (+1, 4000, -3) = (+1, 5230, -3)$ or $1.23 + 4.000 = 5.230$.

Table 5.2 shows, for each operation delivering a result in decimal floating-point format, how the preferred quantum exponents of the operands, $Q(\mathbf{x})$, $Q(\mathbf{y})$, etc., determine the preferred quantum exponent of the operation result, provided the table formula is defined for the arguments. For the cases where the formula is undefined and the function result is $\pm\infty$, the preferred quantum exponent is immaterial because the quantum exponent of $\pm\infty$ is defined to be infinity. For the

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024

other cases where the formula is undefined and the function result is finite, the preferred quantum exponent is unspecified.

NOTE Although unspecified in ISO/IEC 60559, a preferred quantum exponent of 0 for the cases where the formula is undefined and the function result is finite would be a reasonable implementation choice.

Table 5.2 — Preferred quantum exponents

Operation	Preferred quantum exponent of result
`roundeven`, `round`, `trunc`, `ceil`, `floor`, `rint`, `nearbyint`	$\max(Q(\mathbf{x}), 0)$
`nextup`, `nextdown`, `nextafter`, `nexttoward`	least possible
`remainder`	$\min(Q(\mathbf{x}), Q(\mathbf{y}))$
`fmin`, `fmax`, `fminimum`, `fmaximum`, `fminimum_mag`, `fmaximum_mag`, `fminimum_num`, `fmaximum_num`, `fminimum_mag_num`, `fmaximum_mag_num`	$Q(\mathbf{x})$ if **x** gives the result, $Q(\mathbf{y})$ if **y** gives the result
`scalbn`, `scalbln`	$Q(\mathbf{x}) + \mathbf{n}$
`ldexp`	$Q(\mathbf{x}) + \mathbf{p}$
`logb`	0
postfix `++` operator, postfix `--` operator, prefix `++` operator, prefix `--` operator	$\min(Q(\mathbf{x}), 0)$
`+`, `d32add`, `d64add`	$\min(Q(\mathbf{x}), Q(\mathbf{y}))$
`-` , `d32sub`, `d64sub`	$\min(Q(\mathbf{x}), Q(\mathbf{y}))$
`*`, `d32mul`, `d64mul`	$Q(\mathbf{x}) + Q(\mathbf{y})$
`/`, `d32div`, `d64div`	$Q(\mathbf{x}) - Q(\mathbf{y})$
`sqrt`, `d32sqrt`, `d64sqrt`	$\lfloor Q(\mathbf{x})/2 \rfloor$
`fma`, `d32fma`, `d64fma`	$\min(Q(\mathbf{x}) + Q(\mathbf{y}), Q(\mathbf{z}))$
conversion from integer type	0
exact conversion from non-decimal floating type	0
inexact conversion from non-decimal floating type	least possible
conversion between decimal floating types	$Q(\mathbf{x})$
`*cx` returned by `canonicalize`	$Q(*\mathbf{x})$
`strto`, `wcsto`, `scanf`, floating constants of decimal floating type	see 7.24.2.7
`-`(**x**), `+`(**x**)	$Q(\mathbf{x})$
`fabs`	$Q(\mathbf{x})$
`copysign`	$Q(\mathbf{x})$
`quantize`	$Q(\mathbf{y})$
`quantum`	$Q(\mathbf{x})$

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

***encptr** returned by **encodedec**, **encodebin**	$Q(*\mathbf{xptr})$
***xptr** returned by **decodedec**, **decodebin**	$Q(*\mathbf{encptr})$
fmod	$\min(Q(\mathbf{x}), Q(\mathbf{y}))$
fdim	$\min((Q(\mathbf{x}), Q(\mathbf{y}))$ if $\mathbf{x} > \mathbf{y}$, 0 if $\mathbf{x} \leq \mathbf{y}$
cbrt	$\lfloor Q(\mathbf{x})/3 \rfloor$
hypot	$\min(Q(\mathbf{x}), Q(\mathbf{y}))$
pow	$\lfloor \mathbf{y} \times Q(\mathbf{x}) \rfloor$
modf	$Q(\mathbf{value})$
***iptr** returned by **modf**	$\max(Q(\mathbf{value}), 0)$
frexp	$Q(\mathbf{value})$ if $\mathbf{value} = 0$, –(length of coefficient of value) otherwise
***res** returned by **setpayload**, **setpayloadsig**	0 if **pl** does not represent a valid payload, not applicable otherwise (NaN returned)
getpayload	0 if ***x** is a NaN, unspecified otherwise
compoundn	$\lfloor \mathbf{n} \times \min(0, Q(\mathbf{x})) \rfloor$
pown	$\lfloor \mathbf{n} \times Q(\mathbf{x}) \rfloor$
powr	$\lfloor \mathbf{y} \times Q(\mathbf{x}) \rfloor$
rootn	$\lfloor Q(\mathbf{x})/\mathbf{n} \rfloor$
rsqrt	$-\lfloor Q(\mathbf{x})/2 \rfloor$
transcendental functions	0

A function family listed in Table 5.2 indicates the functions for all decimal floating types, where the function family is represented by the name of the functions without a suffix. For example, **ceil** indicates the functions **ceild32**, **ceild64**, and **ceild128**.

Forward references: extended multibyte and wide character utilities <wchar.h> (7.31), floating-point environment <fenv.h> (7.6), general utilities <stdlib.h> (7.24), input/output <stdio.h> (7.23), mathematics <math.h> (7.12), type-generic mathematics <tgmath.h> (7.27), ISO/IEC 60559 floating-point arithmetic (Annex F).

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

6. Language

6.1 Notation

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

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

indicates an optional expression enclosed in braces.

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

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

6.2 Concepts

6.2.1 Scopes of identifiers, type names, and compound literals

An identifier can denote:

— a standard attribute, an attribute prefix, or an attribute name;

— 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.

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.)

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).

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

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub

Full document details: ISO_IEC 9899_2024

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 can 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.

Unless explicitly stated otherwise, where this document uses the term "identifier" to refer to an 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.

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

Structure, union, and enumeration tags have scope that begins just after the appearance of the 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. An ordinary identifier that has an underspecified definition has scope that starts when the definition is completed; if the same ordinary identifier declares another entity with a scope that encloses the current block, that declaration is hidden as soon as the inner declarator is completed.[19] Any other identifier has scope that begins just after the completion of its declarator.

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. A compound literal (which is an expression that provides access to an anonymous object) is associated with the scope of the type name used in its definition; that scope is either file scope, function prototype scope, or block scope.

Forward references: declarations (6.7), function calls (6.5.3.3), function calls (6.5.3.6), function definitions (6.9.2), identifiers (6.4.3), macro replacement (6.10.5), name spaces of identifiers (6.2.3), source file inclusion (6.10.3), statements and blocks (6.8).

6.2.2 Linkages of identifiers

An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.[20] There are three kinds of linkage: external, internal, and none.

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.

If the declaration of a file scope identifier for:

— an object contains any of the storage-class specifiers **`static`** or **`constexpr`**;

— or, a function contains the storage-class specifier **`static`**,

then the identifier has internal linkage.[21]

For an identifier declared with the storage-class specifier **`extern`** in a scope in which a prior declaration of that identifier is visible,[22] 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.

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

[19] That means, that the outer declaration is not visible for the initializer.
[20] There is no linkage between different identifiers.
[21] A function declaration can contain the storage-class specifier **`static`** only if it is at file scope; see 6.7.2.
[22] As specified in 6.2.1, the later declaration can hide the prior declaration.

© ISO/IEC 2024 — All rights reserved

Documents
Document Sharing Hub
Full document details: ISO_IEC 9899_2024