
Programming languages — C

Langages de programmation — C

Reference number
ISO/IEC 9899:1999(E)



PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 1999

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

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 734 10 79
E-mail copyright@iso.ch
Web www.iso.ch

Printed in Switzerland

Contents

Foreword	xi
Introduction	xiv
1. Scope	1
2. Normative references	2
3. Terms, definitions, and symbols	3
4. Conformance	7
5. Environment	9
5.1 Conceptual models	9
5.1.1 Translation environment	9
5.1.2 Execution environments	11
5.2 Environmental considerations	17
5.2.1 Character sets	17
5.2.2 Character display semantics	19
5.2.3 Signals and interrupts	20
5.2.4 Environmental limits	20
6. Language	29
6.1 Notation	29
6.2 Concepts	29
6.2.1 Scopes of identifiers	29
6.2.2 Linkages of identifiers	30
6.2.3 Name spaces of identifiers	31
6.2.4 Storage durations of objects	32
6.2.5 Types	33
6.2.6 Representations of types	37
6.2.7 Compatible type and composite type	40
6.3 Conversions	42
6.3.1 Arithmetic operands	42
6.3.2 Other operands	46
6.4 Lexical elements	49
6.4.1 Keywords	50
6.4.2 Identifiers	51
6.4.3 Universal character names	53
6.4.4 Constants	54
6.4.5 String literals	62
6.4.6 Punctuators	63
6.4.7 Header names	64
6.4.8 Preprocessing numbers	65
6.4.9 Comments	66
6.5 Expressions	67

6.5.1	Primary expressions	69
6.5.2	Postfix operators	69
6.5.3	Unary operators	78
6.5.4	Cast operators	81
6.5.5	Multiplicative operators	82
6.5.6	Additive operators	82
6.5.7	Bitwise shift operators	84
6.5.8	Relational operators	85
6.5.9	Equality operators	86
6.5.10	Bitwise AND operator	87
6.5.11	Bitwise exclusive OR operator	88
6.5.12	Bitwise inclusive OR operator	88
6.5.13	Logical AND operator	89
6.5.14	Logical OR operator	89
6.5.15	Conditional operator	90
6.5.16	Assignment operators	91
6.5.17	Comma operator	94
6.6	Constant expressions	95
6.7	Declarations	97
6.7.1	Storage-class specifiers	98
6.7.2	Type specifiers	99
6.7.3	Type qualifiers	108
6.7.4	Function specifiers	112
6.7.5	Declarators	114
6.7.6	Type names	122
6.7.7	Type definitions	123
6.7.8	Initialization	125
6.8	Statements and blocks	131
6.8.1	Labeled statements	131
6.8.2	Compound statement	132
6.8.3	Expression and null statements	132
6.8.4	Selection statements	133
6.8.5	Iteration statements	135
6.8.6	Jump statements	136
6.9	External definitions	140
6.9.1	Function definitions	141
6.9.2	External object definitions	143
6.10	Preprocessing directives	145
6.10.1	Conditional inclusion	147
6.10.2	Source file inclusion	149
6.10.3	Macro replacement	151
6.10.4	Line control	158
6.10.5	Error directive	159
6.10.6	Pragma directive	159

6.10.7	Null directive	160
6.10.8	Predefined macro names	160
6.10.9	Pragma operator	161
6.11	Future language directions	163
6.11.1	Floating types	163
6.11.2	Linkages of identifiers	163
6.11.3	External names	163
6.11.4	Character escape sequences	163
6.11.5	Storage-class specifiers	163
6.11.6	Function declarators	163
6.11.7	Function definitions	163
6.11.8	Pragma directives	163
6.11.9	Predefined macro names	163
7.	Library	164
7.1	Introduction	164
7.1.1	Definitions of terms	164
7.1.2	Standard headers	165
7.1.3	Reserved identifiers	166
7.1.4	Use of library functions	166
7.2	Diagnostics <assert.h>	169
7.2.1	Program diagnostics	169
7.3	Complex arithmetic <complex.h>	170
7.3.1	Introduction	170
7.3.2	Conventions	171
7.3.3	Branch cuts	171
7.3.4	The CX_LIMITED_RANGE pragma	171
7.3.5	Trigonometric functions	172
7.3.6	Hyperbolic functions	174
7.3.7	Exponential and logarithmic functions	176
7.3.8	Power and absolute-value functions	177
7.3.9	Manipulation functions	178
7.4	Character handling <ctype.h>	181
7.4.1	Character classification functions	181
7.4.2	Character case mapping functions	184
7.5	Errors <errno.h>	186
7.6	Floating-point environment <fenv.h>	187
7.6.1	The FENV_ACCESS pragma	189
7.6.2	Floating-point exceptions	190
7.6.3	Rounding	192
7.6.4	Environment	194
7.7	Characteristics of floating types <float.h>	196
7.8	Format conversion of integer types <inttypes.h>	197
7.8.1	Macros for format specifiers	197
7.8.2	Functions for greatest-width integer types	198

7.9	Alternative spellings <iso646.h>	201
7.10	Sizes of integer types <limits.h>	202
7.11	Localization <locale.h>	203
7.11.1	Locale control	204
7.11.2	Numeric formatting convention inquiry	205
7.12	Mathematics <math.h>	211
7.12.1	Treatment of error conditions	213
7.12.2	The FP_CONTRACT pragma	214
7.12.3	Classification macros	215
7.12.4	Trigonometric functions	217
7.12.5	Hyperbolic functions	220
7.12.6	Exponential and logarithmic functions	222
7.12.7	Power and absolute-value functions	227
7.12.8	Error and gamma functions	229
7.12.9	Nearest integer functions	230
7.12.10	Remainder functions	234
7.12.11	Manipulation functions	235
7.12.12	Maximum, minimum, and positive difference functions	237
7.12.13	Floating multiply-add	238
7.12.14	Comparison macros	239
7.13	Nonlocal jumps <setjmp.h>	242
7.13.1	Save calling environment	242
7.13.2	Restore calling environment	243
7.14	Signal handling <signal.h>	245
7.14.1	Specify signal handling	246
7.14.2	Send signal	247
7.15	Variable arguments <stdarg.h>	248
7.15.1	Variable argument list access macros	248
7.16	Boolean type and values <stdbool.h>	252
7.17	Common definitions <stddef.h>	253
7.18	Integer types <stdint.h>	254
7.18.1	Integer types	254
7.18.2	Limits of specified-width integer types	256
7.18.3	Limits of other integer types	258
7.18.4	Macros for integer constants	259
7.19	Input/output <stdio.h>	261
7.19.1	Introduction	261
7.19.2	Streams	263
7.19.3	Files	265
7.19.4	Operations on files	267
7.19.5	File access functions	269
7.19.6	Formatted input/output functions	273
7.19.7	Character input/output functions	294
7.19.8	Direct input/output functions	299

7.19.9	File positioning functions	300
7.19.10	Error-handling functions	303
7.20	General utilities <stdlib.h>	305
7.20.1	Numeric conversion functions	306
7.20.2	Pseudo-random sequence generation functions	311
7.20.3	Memory management functions	312
7.20.4	Communication with the environment	314
7.20.5	Searching and sorting utilities	317
7.20.6	Integer arithmetic functions	319
7.20.7	Multibyte/wide character conversion functions	320
7.20.8	Multibyte/wide string conversion functions	322
7.21	String handling <string.h>	324
7.21.1	String function conventions	324
7.21.2	Copying functions	324
7.21.3	Concatenation functions	326
7.21.4	Comparison functions	327
7.21.5	Search functions	329
7.21.6	Miscellaneous functions	332
7.22	Type-generic math <tgmath.h>	334
7.23	Date and time <time.h>	337
7.23.1	Components of time	337
7.23.2	Time manipulation functions	338
7.23.3	Time conversion functions	340
7.24	Extended multibyte and wide character utilities <wchar.h>	347
7.24.1	Introduction	347
7.24.2	Formatted wide character input/output functions	348
7.24.3	Wide character input/output functions	366
7.24.4	General wide string utilities	370
7.24.5	Wide character time conversion functions	384
7.24.6	Extended multibyte/wide character conversion utilities	385
7.25	Wide character classification and mapping utilities <wctype.h>	392
7.25.1	Introduction	392
7.25.2	Wide character classification utilities	393
7.25.3	Wide character case mapping utilities	398
7.26	Future library directions	400
7.26.1	Complex arithmetic <complex.h>	400
7.26.2	Character handling <ctype.h>	400
7.26.3	Errors <errno.h>	400
7.26.4	Format conversion of integer types <inttypes.h>	400
7.26.5	Localization <locale.h>	400
7.26.6	Signal handling <signal.h>	400
7.26.7	Boolean type and values <stdbool.h>	400
7.26.8	Integer types <stdint.h>	400
7.26.9	Input/output <stdio.h>	401

7.26.10	General utilities <stdlib.h>	401
7.26.11	String handling <string.h>	401
7.26.12	Extended multibyte and wide character utilities <wchar.h>	401
7.26.13	Wide character classification and mapping utilities <wctype.h>	401
Annex A	(informative) Language syntax summary	402
A.1	Lexical grammar	402
A.2	Phrase structure grammar	408
A.3	Preprocessing directives	415
Annex B	(informative) Library summary	417
B.1	Diagnostics <assert.h>	417
B.2	Complex <complex.h>	417
B.3	Character handling <ctype.h>	419
B.4	Errors <errno.h>	419
B.5	Floating-point environment <fenv.h>	419
B.6	Characteristics of floating types <float.h>	420
B.7	Format conversion of integer types <inttypes.h>	420
B.8	Alternative spellings <iso646.h>	421
B.9	Sizes of integer types <limits.h>	421
B.10	Localization <locale.h>	421
B.11	Mathematics <math.h>	421
B.12	Nonlocal jumps <setjmp.h>	426
B.13	Signal handling <signal.h>	426
B.14	Variable arguments <stdarg.h>	426
B.15	Boolean type and values <stdbool.h>	426
B.16	Common definitions <stddef.h>	427
B.17	Integer types <stdint.h>	427
B.18	Input/output <stdio.h>	427
B.19	General utilities <stdlib.h>	429
B.20	String handling <string.h>	431
B.21	Type-generic math <tgmath.h>	432
B.22	Date and time <time.h>	432
B.23	Extended multibyte/wide character utilities <wchar.h>	433
B.24	Wide character classification and mapping utilities <wctype.h>	435
Annex C	(informative) Sequence points	437
Annex D	(normative) Universal character names for identifiers	438
Annex E	(informative) Implementation limits	440
Annex F	(normative) IEC 60559 floating-point arithmetic	442
F.1	Introduction	442
F.2	Types	442
F.3	Operators and functions	443

F.4	Floating to integer conversion	445
F.5	Binary-decimal conversion	445
F.6	Contracted expressions	446
F.7	Floating-point environment	446
F.8	Optimization	449
F.9	Mathematics <math.h>	452
Annex G (informative)	IEC 60559-compatible complex arithmetic	465
G.1	Introduction	465
G.2	Types	465
G.3	Conventions	465
G.4	Conversions	466
G.5	Binary operators	466
G.6	Complex arithmetic <complex.h>	470
G.7	Type-generic math <tgmath.h>	478
Annex H (informative)	Language independent arithmetic	479
H.1	Introduction	479
H.2	Types	479
H.3	Notification	483
Annex I (informative)	Common warnings	485
Annex J (informative)	Portability issues	487
J.1	Unspecified behavior	487
J.2	Undefined behavior	490
J.3	Implementation-defined behavior	503
J.4	Locale-specific behavior	510
J.5	Common extensions	511
Bibliography		514
Index		517

Foreword

- 1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are member of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.
- 2 International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.
- 3 In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.
- 4 International Standard ISO/IEC 9899 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*. The Working Group responsible for this standard (WG 14) maintains a site on the World Wide Web at <http://www.dkuug.dk/JTC1/SC22/WG14/> containing additional information relevant to this standard such as a Rationale for many of the decisions made during its preparation and a log of Defect Reports and Responses.
- 5 This second edition cancels and replaces the first edition, ISO/IEC 9899:1990, as amended and corrected by ISO/IEC 9899/COR1:1994, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996. Major changes from the previous edition include:
 - restricted character set support via digraphs and **<iso646.h>** (originally specified in AMD1)
 - wide character library support in **<wchar.h>** and **<wctype.h>** (originally specified in AMD1)
 - more precise aliasing rules via effective type
 - restricted pointers
 - variable-length arrays
 - flexible array members
 - **static** and type qualifiers in parameter array declarators
 - complex (and imaginary) support in **<complex.h>**
 - type-generic math macros in **<tgmath.h>**

- the **long long int** type and library functions
- increased minimum translation limits
- additional floating-point characteristics in **<float.h>**
- remove implicit **int**
- reliable integer division
- universal character names (**\u** and **\U**)
- extended identifiers
- hexadecimal floating-point constants and **%a** and **%A printf/scanf** conversion specifiers
- compound literals
- designated initializers
- **//** comments
- extended integer types and library functions in **<inttypes.h>** and **<stdint.h>**
- remove implicit function declaration
- preprocessor arithmetic done in **intmax_t/uintmax_t**
- mixed declarations and code
- new block scopes for selection and iteration statements
- integer constant type rules
- integer promotion rules
- macros with a variable number of arguments
- the **vscanf** family of functions in **<stdio.h>** and **<wchar.h>**
- additional math library functions in **<math.h>**
- floating-point environment access in **<fenv.h>**
- IEC 60559 (also known as IEC 559 or IEEE arithmetic) support
- trailing comma allowed in **enum** declaration
- **%lf** conversion specifier allowed in **printf**
- inline functions
- the **snprintf** family of functions in **<stdio.h>**
- boolean type in **<stdbool.h>**
- idempotent type qualifiers
- empty macro arguments

- new struct type compatibility rules (tag compatibility)
 - additional predefined macro names
 - **_Pragma** preprocessing operator
 - standard pragmas
 - **__func__** predefined identifier
 - **VA_COPY** macro
 - additional **strftime** conversion specifiers
 - LIA compatibility annex
 - deprecate **ungetc** at the beginning of a binary file
 - remove deprecation of aliased array parameters
 - conversion of array to pointer not limited to lvalues
 - relaxed constraints on aggregate and union initialization
 - relaxed restrictions on portable header names
 - **return** without expression not permitted in function that returns a value (and vice versa)
- 6 Annexes D and F form a normative part of this standard; annexes A, B, C, E, G, H, I, J, the bibliography, and the index are for information only. In accordance with Part 3 of the ISO/IEC Directives, this foreword, the introduction, notes, footnotes, and examples are also for information only.

Introduction

- 1 With the introduction of new devices and extended character sets, new features may be added to this International Standard. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.
- 2 Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this International Standard. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.26]) is discouraged.
- 3 This International Standard is divided into four major subdivisions:
 - preliminary elements (clauses 1–4);
 - the characteristics of environments that translate and execute C programs (clause 5);
 - the language syntax, constraints, and semantics (clause 6);
 - the library facilities (clause 7).
- 4 Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this International Standard. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes provide additional information and summarize the information contained in this International Standard. A bibliography lists documents that were referred to during the preparation of the standard.
- 5 The language clause (clause 6) is derived from “The C Reference Manual”.
- 6 The library clause (clause 7) is based on the *1984 /usr/group Standard*.

Programming languages — C

1. Scope

- 1 This International Standard specifies the form and establishes the interpretation of programs written in the C programming language.¹⁾ It specifies
 - the representation of C programs;
 - the syntax and constraints of the C language;
 - the semantic rules for interpreting C programs;
 - the representation of input data to be processed by C programs;
 - the representation of output data produced by C programs;
 - the restrictions and limits imposed by a conforming implementation of C.
- 2 This International Standard does not specify
 - the mechanism by which C programs are transformed for use by a data-processing system;
 - the mechanism by which C programs are invoked for use by a data-processing system;
 - the mechanism by which input data are transformed for use by a C program;
 - the mechanism by which output data are transformed after being produced by a C program;

1) This International Standard is designed to promote the portability of C programs among a variety of data-processing systems. It is intended for use by implementors and programmers.

- 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.

2. Normative references

- 1 The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.
- 2 ISO 31-11:1992, *Quantities and units — Part 11: Mathematical signs and symbols for use in the physical sciences and technology*.
- 3 ISO/IEC 646, *Information technology — ISO 7-bit coded character set for information interchange*.
- 4 ISO/IEC 2382-1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*.
- 5 ISO 4217, *Codes for the representation of currencies and funds*.
- 6 ISO 8601, *Data elements and interchange formats — Information interchange — Representation of dates and times*.
- 7 ISO/IEC 10646 (all parts), *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*.
- 8 IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989).

3. Terms, definitions, and symbols

- 1 For the purposes of this International Standard, the following definitions apply. Other terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard are to be interpreted according to ISO/IEC 2382-1. Mathematical symbols not defined in this International Standard are to be interpreted according to ISO 31-11.

3.1

1 **access**

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

- 2 NOTE 1 Where only one of these two actions is meant, “read” or “modify” is used.

- 3 NOTE 2 “Modify” includes the case where the new value being stored is the same as the previous value.

- 4 NOTE 3 Expressions that are not evaluated do not access objects.

3.2

1 **alignment**

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

3.3

1 **argument**

actual argument

actual parameter (deprecated)

expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

3.4

1 **behavior**

external appearance or action

3.4.1

1 **implementation-defined behavior**

unspecified behavior where each implementation documents how the choice is made

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

3.4.2

1 **locale-specific behavior**

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

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

3.4.3

1 undefined behavior

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

- 2 NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

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

3.4.4

1 unspecified behavior

behavior where this International Standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance

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

3.5

1 bit

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

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

3.6

1 byte

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

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

- 3 NOTE 2 A byte is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*.

3.7

1 character

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

3.7.1

1 character

single-byte character

⟨C⟩ bit representation that fits in a byte

3.7.2

1 **multibyte character**

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

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

3.7.3

1 **wide character**

bit representation that fits in an object of type **wchar_t**, capable of representing any character in the current locale

3.8

1 **constraint**

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

3.9

1 **correctly rounded result**

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

3.10

1 **diagnostic message**

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

3.11

1 **forward reference**

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

3.12

1 **implementation**

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

3.13

1 **implementation limit**

restriction imposed upon programs by the implementation

3.14

1 **object**

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

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

3.15

1 **parameter**

formal parameter

formal argument (deprecated)

object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition

3.16

1 **recommended practice**

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

3.17

1 **value**

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

3.17.1

1 **implementation-defined value**

unspecified value where each implementation documents how the choice is made

3.17.2

1 **indeterminate value**

either an unspecified value or a trap representation

3.17.3

1 **unspecified value**

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

2 NOTE An unspecified value cannot be a trap representation.

3.18

1 $\lceil x \rceil$

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

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

3.19

1 $\lfloor x \rfloor$

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

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

4. Conformance

- 1 In this International Standard, “shall” is to be interpreted as a requirement on an implementation or on a program; conversely, “shall not” is to be interpreted as a prohibition.
- 2 If a “shall” or “shall not” requirement that appears outside of a constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in this International Standard by the words “undefined behavior” or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe “behavior that is undefined”.
- 3 A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3.
- 4 The implementation shall not successfully translate a preprocessing translation unit containing a **#error** preprocessing directive unless it is part of a group skipped by conditional inclusion.
- 5 A *strictly conforming program* shall use only those features of the language and library specified in this International Standard.²⁾ It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit.
- 6 The two forms of *conforming implementation* are hosted and freestanding. A *conforming hosted implementation* shall accept any strictly conforming program. A *conforming freestanding implementation* shall accept any strictly conforming program that does not use complex types and 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>**, **<stdarg.h>**, **<stdbool.h>**, **<stddef.h>**, and **<stdint.h>**. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.³⁾

-
- 2) A strictly conforming program can use conditional features (such as those in annex F) provided the use is guarded by a **#ifdef** directive with the appropriate macro. For example:

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

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

- 7 A *conforming program* is one that is acceptable to a conforming implementation.⁴⁾
- 8 An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions.

Forward references: conditional inclusion (6.10.1), error directive (6.10.5), characteristics of floating types **<float.h>** (7.7), alternative spellings **<iso646.h>** (7.9), sizes of integer types **<limits.h>** (7.10), variable arguments **<stdarg.h>** (7.15), boolean type and values **<stdbool.h>** (7.16), common definitions **<stddef.h>** (7.17), integer types **<stdint.h>** (7.18).

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

5. Environment

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

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

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Program structure

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

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

5.1.1.2 Translation phases

- 1 The precedence among the syntax rules of translation is specified by the following phases.⁵⁾
 1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary. Trigraph sequences are replaced by corresponding single-character internal representations.
 2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part

5) Implementations shall behave as if these separate phases occur, even though many are typically folded together in practice.

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 tokens⁶⁾ and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment. Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.
4. Preprocessing directives are executed, macro invocations are expanded, and **_Pragma** unary operator expressions are executed. If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined. A **#include** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.
5. Each source character set member and escape sequence in character constants and string literals is converted to the corresponding member of the execution character set; if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.⁷⁾
6. Adjacent string literal tokens are concatenated.
7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated as a translation unit.
8. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

Forward references: universal character names (6.4.3), lexical elements (6.4), preprocessing directives (6.10), trigraph sequences (5.2.1.1), external definitions (6.9).

6) 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.

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

5.1.1.3 Diagnostics

- 1 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. Diagnostic messages need not be produced in other circumstances.⁸⁾
- 2 **EXAMPLE** An implementation shall issue a diagnostic for the translation unit:

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

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

5.1.2 Execution environments

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

Forward references: storage durations of objects (6.2.4), initialization (6.7.8).

5.1.2.1 Freestanding environment

- 1 In a freestanding environment (in which C program execution may take place without any benefit of an operating system), the name and type of the function called at program startup are implementation-defined. Any library facilities available to a freestanding program, other than the minimal set required by clause 4, are implementation-defined.
- 2 The effect of program termination in a freestanding environment is implementation-defined.

5.1.2.2 Hosted environment

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

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

5.1.2.2.1 Program startup

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

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

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

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

or equivalent;⁹⁾ or in some other implementation-defined manner.

- 2 If they are declared, the parameters to the **main** function shall obey the following constraints:
 - The value of **argc** shall be nonnegative.
 - **argv[argc]** shall be a null pointer.
 - If the value of **argc** is greater than zero, the array members **argv[0]** through **argv[argc-1]** inclusive shall contain pointers to strings, which are given implementation-defined values by the host environment prior to program startup. The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase.
 - If the value of **argc** is greater than zero, the string pointed to by **argv[0]** represents the *program name*; **argv[0][0]** shall be the null character if the program name is not available from the host environment. If the value of **argc** is greater than one, the strings pointed to by **argv[1]** through **argv[argc-1]** represent the *program parameters*.
 - The parameters **argc** and **argv** and the strings pointed to by the **argv** array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

5.1.2.2.2 Program execution

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

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

5.1.2.2.3 Program termination

- 1 If the return type of the **main** function is a type compatible with **int**, a return from the initial call to the **main** function is equivalent to calling the **exit** function with the value returned by the **main** function as its argument;¹⁰⁾ 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 (7.20.4.3).

5.1.2.3 Program execution

- 1 The semantic descriptions in this International Standard describe the behavior of an abstract machine in which issues of optimization are irrelevant.
- 2 Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*,¹¹⁾ which are changes in the state of the execution environment. Evaluation of an expression may produce side effects. At certain specified points in the execution sequence called *sequence points*, all side effects of previous evaluations shall be complete and no side effects of subsequent evaluations shall have taken place. (A summary of the sequence points is given in annex C.)
- 3 In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).
- 4 When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet.
- 5 The least requirements on a conforming implementation are:
 - At sequence points, volatile objects are stable in the sense that previous accesses are complete and subsequent accesses have not yet occurred.

10) 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.

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

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

- 6 What constitutes an interactive device is implementation-defined.
- 7 More stringent correspondences between abstract and actual semantics may be defined by each implementation.
- 8 **EXAMPLE 1** An implementation might define a one-to-one correspondence between abstract and actual semantics: at every sequence point, the values of the actual objects would agree with those specified by the abstract semantics. The keyword **volatile** would then be redundant.
- 9 Alternatively, an implementation might perform various optimizations within each translation unit, such that the actual semantics would agree with the abstract semantics only when making function calls across translation unit boundaries. In such an implementation, at the time of each function entry and function return where the calling function and the called function are in different translation units, the values of all externally linked objects and of all objects accessible via pointers therein would agree with the abstract semantics. Furthermore, at the time of each such function entry the values of the parameters of the called function and of all objects accessible via pointers therein would agree with the abstract semantics. In this type of implementation, objects referred to by interrupt service routines activated by the **signal** function would require explicit specification of **volatile** storage, as well as other implementation-defined restrictions.

- 10 **EXAMPLE 2** In executing the fragment

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

the “integer promotions” require that the abstract machine promote the value of each variable to **int** size and then add the two **ints** and truncate the sum. Provided the addition of two **chars** can be done without overflow, or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

- 11 **EXAMPLE 3** Similarly, in the fragment

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

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

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

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

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

- 13 EXAMPLE 5 Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real numbers are often not valid (see F.8).

```
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;
```

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

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

the expression statement behaves exactly the same as

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

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

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

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

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

or

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

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

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

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

the expression statement is grouped as if it were written as

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

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

Forward references: expressions (6.5), type qualifiers (6.7.3), statements (6.8), the **signal** function (7.14), files (7.19.3).

5.2 Environmental considerations

5.2.1 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

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

the 26 *lowercase letters* of the Latin alphabet

a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z

the 10 decimal *digits*

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

the following 29 graphic characters

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

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

converted to a token), the behavior is undefined.

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

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

5.2.1.1 Trigraph sequences

- 1 All occurrences in a source file of the following sequences of three characters (called *trigraph sequences*¹²⁾) are replaced with the corresponding single character.

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

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

- 2 **EXAMPLE** The following source line

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

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

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

5.2.1.2 Multibyte characters

- 1 The source character set may contain multibyte characters, used to represent members of the extended character set. The execution character set may also contain multibyte characters, which need not have the same encoding as for the source character set. For both character sets, the following shall hold:
 - The basic character set shall be present and each character shall be encoded as a single byte.
 - The presence, meaning, and representation of any additional members is locale-specific.
 - A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states* when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes

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

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.
 - A byte with all bits zero shall not occur in the second or subsequent bytes of a multibyte character.
- 2 For source files, the following shall hold:
- An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state.
 - An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

5.2.2 Character display semantics

- 1 The *active position* is that location on a display device where the next character output by the **fputc** function would appear. The intent of writing a printing character (as defined by the **isprint** function) to a display device is to display a graphic representation of that character at the active position and then advance the active position to the next position on the current line. The direction of writing is locale-specific. If the active position is at the final position of a line (if there is one), the behavior of the display device is unspecified.
- 2 Alphabetic escape sequences representing nongraphic characters in the execution character set are intended to produce actions on display devices as follows:
- \a** (*alert*) Produces an audible or visible alert without changing the active position.
 - \b** (*backspace*) Moves the active position to the previous position on the current line. If the active position is at the initial position of a line, the behavior of the display device is unspecified.
 - \f** (*form feed*) Moves the active position to the initial position at the start of the next logical page.
 - \n** (*new line*) Moves the active position to the initial position of the next line.
 - \r** (*carriage return*) Moves the active position to the initial position of the current line.
 - \t** (*horizontal tab*) Moves the active position to the next horizontal tabulation position on the current line. If the active position is at or past the last defined horizontal tabulation position, the behavior of the display device is unspecified.
 - \v** (*vertical tab*) Moves the active position to the initial position of the next vertical tabulation position. If the active position is at or past the last defined vertical tabulation position, the behavior of the display device is unspecified.

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

Forward references: the **isprint** function (7.4.1.8), the **fputc** function (7.19.7.3).

5.2.3 Signals and interrupts

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

5.2.4 Environmental limits

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

5.2.4.1 Translation limits

- 1 The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits: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 incomplete type in a declaration
 - 63 nesting levels of parenthesized declarators within a full declarator
 - 63 nesting levels of parenthesized expressions within a full expression
 - 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character)
 - 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same

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

- 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 character string literal or wide string literal (after concatenation)
 - 65535 bytes in an object (in a hosted environment only)
 - 15 nesting levels for **#included** files
 - 1023 **case** labels for a **switch** statement (excluding those for any nested **switch** statements)
 - 1023 members in a single structure or union
 - 1023 enumeration constants in a single enumeration
 - 63 levels of nested structure or union definitions in a single struct-declaration-list

5.2.4.2 Numerical limits

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

Forward references: integer types **<stdint.h>** (7.18).

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

- 1 The values given below shall be replaced by constant expressions suitable for use in **#if** preprocessing directives. Moreover, except for **CHAR_BIT** and **MB_LEN_MAX**, the following shall be replaced by expressions that have the same type as would an expression that is an object of the corresponding type converted according to the integer promotions. Their implementation-defined values shall be equal or greater in magnitude (absolute value) to those shown, with the same sign.

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

- number of bits for smallest object that is not a bit-field (byte)
CHAR_BIT **8**
- minimum value for an object of type **signed char**
SCHAR_MIN **-127 // $-(2^7 - 1)$**
- maximum value for an object of type **signed char**
SCHAR_MAX **+127 // $2^7 - 1$**
- maximum value for an object of type **unsigned char**
UCHAR_MAX **255 // $2^8 - 1$**
- minimum value for an object of type **char**
CHAR_MIN *see below*
- maximum value for an object of type **char**
CHAR_MAX *see below*
- maximum number of bytes in a multibyte character, for any supported locale
MB_LEN_MAX **1**
- minimum value for an object of type **short int**
SHRT_MIN **-32767 // $-(2^{15} - 1)$**
- maximum value for an object of type **short int**
SHRT_MAX **+32767 // $2^{15} - 1$**
- maximum value for an object of type **unsigned short int**
USHRT_MAX **65535 // $2^{16} - 1$**
- minimum value for an object of type **int**
INT_MIN **-32767 // $-(2^{15} - 1)$**
- maximum value for an object of type **int**
INT_MAX **+32767 // $2^{15} - 1$**
- maximum value for an object of type **unsigned int**
UINT_MAX **65535 // $2^{16} - 1$**
- minimum value for an object of type **long int**
LONG_MIN **-2147483647 // $-(2^{31} - 1)$**
- maximum value for an object of type **long int**
LONG_MAX **+2147483647 // $2^{31} - 1$**
- maximum value for an object of type **unsigned long int**
ULONG_MAX **4294967295 // $2^{32} - 1$**
- minimum value for an object of type **long long int**
LLONG_MIN **-9223372036854775807 // $-(2^{63} - 1)$**

— maximum value for an object of type **long long int**
LLONG_MAX **+9223372036854775807** // $2^{63} - 1$

— maximum value for an object of type **unsigned long long int**
ULLONG_MAX **18446744073709551615** // $2^{64} - 1$

- 2 If the value of an object of type **char** is treated as a signed integer when used in an expression, the value of **CHAR_MIN** shall be the same as that of **SCHAR_MIN** and the value of **CHAR_MAX** shall be the same as that of **SCHAR_MAX**. Otherwise, the value of **CHAR_MIN** shall be 0 and the value of **CHAR_MAX** shall be the same as that of **UCHAR_MAX**.¹⁵⁾ The value **UCHAR_MAX** shall equal $2^{\text{CHAR_BIT}} - 1$.

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

5.2.4.2.2 Characteristics of floating types <float.h>

- 1 The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.¹⁶⁾ The following parameters are used to define the model for each floating-point type:

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

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

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

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

¹⁵⁾ See 6.2.5.

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.

arithmetic operand.¹⁷⁾

- 4 The accuracy of the floating-point operations (+, -, *, /) and of the library functions in **<math.h>** and **<complex.h>** that return floating-point results is implementation-defined. The implementation may state that the accuracy is unknown.
- 5 All integer values in the **<float.h>** header, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives; all floating values shall be constant expressions. All except **DECIMAL_DIG**, **FLT_EVAL_METHOD**, **FLT_RADIX**, and **FLT_ROUNDS** have separate names for all three floating-point types. The floating-point model representation is provided for all values except **FLT_EVAL_METHOD** and **FLT_ROUNDS**.
- 6 The rounding mode for floating-point addition is characterized by the implementation-defined value of **FLT_ROUNDS**:¹⁸⁾
 - 1 indeterminate
 - 0 toward zero
 - 1 to nearest
 - 2 toward positive infinity
 - 3 toward negative infinity

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

- 7 The values of operations with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type. The use of evaluation formats is characterized by the implementation-defined value of **FLT_EVAL_METHOD**:¹⁹⁾
 - 1 indeterminate;
 - 0 evaluate all operations and constants just to the range and precision of the type;

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

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

19) 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**.

- 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.

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

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

DECIMAL_DIG **10**

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

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

FLT_DIG **6**
DBL_DIG **10**
LDBL_DIG **10**

- ```
FLT_MIN_EXP
DBL_MIN_EXP
LDBL_MIN_EXP
```

- ```

FLT_MIN_10_EXP          -37
DBL_MIN_10_EXP          -37
LDBL_MIN_10_EXP         -37

```

- ```
FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP
```

- ```

FLT_MAX_10_EXP          +37
DBL_MAX_10_EXP          +37
LDBL_MAX_10_EXP         +37

```

- ```

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

```

- ```

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

```

— minimum normalized positive floating-point number, $b^{e_{\min}-1}$

FLT_MIN	1E-37
DBL_MIN	1E-37
LDBL_MIN	1E-37

Recommended practice

- 11 Conversion from (at least) **double** to decimal with **DECIMAL_DIG** digits and back should be the identity function.
- 12 EXAMPLE 1 The following describes an artificial floating-point representation that meets the minimum requirements of this International Standard, and the appropriate values in a **<float.h>** header for type **float**:

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

FLT_RADIX	16
FLT_MANT_DIG	6
FLT_EPSILON	9.53674316E-07F
FLT_DIG	6
FLT_MIN_EXP	-31
FLT_MIN	2.93873588E-39F
FLT_MIN_10_EXP	-38
FLT_MAX_EXP	+32
FLT_MAX	3.40282347E+38F
FLT_MAX_10_EXP	+38

- 13 EXAMPLE 2 The following describes floating-point representations that also meet the requirements for single-precision and double-precision normalized numbers in IEC 60559,²⁰⁾ and the appropriate values in a **<float.h>** header for types **float** and **double**:

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

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

FLT_RADIX	2
DECIMAL_DIG	17
FLT_MANT_DIG	24
FLT_EPSILON	1.19209290E-07F // decimal constant
FLT_EPSILON	0X1P-23F // hex constant
FLT_DIG	6
FLT_MIN_EXP	-125
FLT_MIN	1.17549435E-38F // decimal constant
FLT_MIN	0X1P-126F // hex constant

20) The floating-point model in that standard sums powers of b from zero, so the values of the exponent limits are one less than shown here.

```

FLT_MIN_10_EXP          -37
FLT_MAX_EXP             +128
FLT_MAX                 3.40282347E+38F // decimal constant
FLT_MAX                 0X1.fffffeP127F // hex constant
FLT_MAX_10_EXP          +38
DBL_MANT_DIG             53
DBL_EPSILON             2.2204460492503131E-16 // decimal constant
DBL_EPSILON             0X1P-52 // hex constant
DBL_DIG                 15
DBL_MIN_EXP             -1021
DBL_MIN                 2.2250738585072014E-308 // decimal constant
DBL_MIN                 0X1P-1022 // hex constant
DBL_MIN_10_EXP          -307
DBL_MAX_EXP             +1024
DBL_MAX                 1.7976931348623157E+308 // decimal constant
DBL_MAX                 0X1.fffffffffffffP1023 // hex constant
DBL_MAX_10_EXP          +308

```

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

Forward references: conditional inclusion (6.10.1), complex arithmetic **<complex.h>** (7.3), floating-point environment **<fenv.h>** (7.6), mathematics **<math.h>** (7.12).

6. Language

6.1 Notation

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

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

indicates an optional expression enclosed in braces.

- 2 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.
- 3 A summary of the language syntax is given in annex A.

6.2 Concepts

6.2.1 Scopes of identifiers

- 1 An identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. The same identifier can denote different entities at different points in the program. A member of an enumeration is called an *enumeration constant*. Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.
- 2 For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. Different entities designated by the same identifier either have different scopes, or are in different name spaces. There are four kinds of scopes: function, file, block, and function prototype. (A *function prototype* is a declaration of a function that declares the types of its parameters.)
- 3 A label name is the only kind of identifier that has *function scope*. It can be used (in a **goto** statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a : and a statement).
- 4 Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier). If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit. If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block. If the declarator or type specifier that declares the identifier appears

within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator. If an identifier designates two different entities in the same name space, the scopes might overlap. If so, the scope of one entity (the *inner scope*) will be a strict subset of the scope of the other entity (the *outer scope*). Within the inner scope, the identifier designates the entity declared in the inner scope; the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

- 5 Unless explicitly stated otherwise, where this International Standard uses the term “identifier” to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.
- 6 Two identifiers have the *same scope* if and only if their scopes terminate at the same point.
- 7 Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has scope that begins just after the completion of its declarator.

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

6.2.2 Linkages of identifiers

- 1 An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.²¹⁾ There are three kinds of linkage: external, internal, and none.
- 2 In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. Within one translation unit, each declaration of an identifier with *internal linkage* denotes the same object or function. Each declaration of an identifier with *no linkage* denotes a unique entity.
- 3 If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **static**, the identifier has internal linkage.²²⁾
- 4 For an identifier declared with the storage-class specifier **extern** in a scope in which a

21) There is no linkage between different identifiers.

22) A function declaration can contain the storage-class specifier **static** only if it is at file scope; see 6.7.1.

prior declaration of that identifier is visible,²³⁾ if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

- 5 If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **extern**. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.
- 6 The following identifiers have no linkage: an identifier declared to be anything other than an object or a function; an identifier declared to be a function parameter; a block scope identifier for an object declared without the storage-class specifier **extern**.
- 7 If, within a translation unit, the same identifier appears with both internal and external linkage, the behavior is undefined.

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

6.2.3 Name spaces of identifiers

- 1 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:
 - *label names* (disambiguated by the syntax of the label declaration and use);
 - the *tags* of structures, unions, and enumerations (disambiguated by following any²⁴⁾ of the keywords **struct**, **union**, or **enum**);
 - the *members* of structures or unions; each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the **.** or **->** operator);
 - all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

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

23) As specified in 6.2.1, the later declaration might hide the prior declaration.

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

6.2.4 Storage durations of objects

- 1 An object has a *storage duration* that determines its lifetime. There are three storage durations: static, automatic, and allocated. Allocated storage is described in 7.20.3.
- 2 The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address,²⁵⁾ and retains its last-stored value throughout its lifetime.²⁶⁾ If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.
- 3 An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **static** has *static storage duration*. Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.
- 4 An object whose identifier is declared with no linkage and without the storage-class specifier **static** has *automatic storage duration*.
- 5 For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) If the block is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate. If an initialization is specified for the object, it is performed each time the declaration is reached in the execution of the block; otherwise, the value becomes indeterminate each time the declaration is reached.
- 6 For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.²⁷⁾ If the scope is entered recursively, a new instance of the object is created each time. The initial value of the object is indeterminate.

Forward references: statements (6.8), function calls (6.5.2.2), declarators (6.7.5), array declarators (6.7.5.2), initialization (6.7.8).

25) The term “constant address” means that two pointers to the object constructed at possibly different times will compare equal. The address may be different during two different executions of the same program.

26) In the case of a volatile object, the last store need not be explicit in the program.

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

6.2.5 Types

- 1 The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that fully describe objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes).
- 2 An object declared as type **_Bool** is large enough to store the values 0 and 1.
- 3 An object declared as type **char** is large enough to store any member of the basic execution character set. If a member of the basic execution character set is stored in a **char** object, its value is guaranteed to be positive. If any other character is stored in a **char** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type.
- 4 There are five *standard signed integer types*, designated as **signed char**, **short int**, **int**, **long int**, and **long long int**. (These and other types may be designated in several additional ways, as described in 6.7.2.) There may also be implementation-defined *extended signed integer types*.²⁸⁾ The standard and extended signed integer types are collectively called *signed integer types*.²⁹⁾
- 5 An object declared as type **signed char** occupies the same amount of storage as a “plain” **char** object. A “plain” **int** object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range **INT_MIN** to **INT_MAX** as defined in the header **<limits.h>**).
- 6 For each of the signed integer types, there is a corresponding (but different) unsigned integer type (designated with the keyword **unsigned**) that uses the same amount of storage (including sign information) and has the same alignment requirements. The type **_Bool** and the unsigned integer types that correspond to the standard signed integer types are the *standard unsigned integer types*. The unsigned integer types that correspond to the extended signed integer types are the *extended unsigned integer types*. The standard and extended unsigned integer types are collectively called *unsigned integer types*.³⁰⁾

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

29) Therefore, any statement in this Standard about signed integer types also applies to the extended signed integer types.

30) Therefore, any statement in this Standard about unsigned integer types also applies to the extended unsigned integer types.

- 7 The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*, the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.
- 8 For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type.
- 9 The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the representation of the same value in each type is the same.³¹⁾ A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type.
- 10 There are three *real floating types*, designated as **float**, **double**, and **long double**.³²⁾ The set of values of the type **float** is a subset of the set of values of the type **double**; the set of values of the type **double** is a subset of the set of values of the type **long double**.
- 11 There are three *complex types*, designated as **float _Complex**, **double _Complex**, and **long double _Complex**.³³⁾ The real floating and complex types are collectively called the *floating types*.
- 12 For each floating type there is a *corresponding real type*, which is always a real floating type. For real floating types, it is the same type. For complex types, it is the type given by deleting the keyword **_Complex** from the type name.
- 13 Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; the first element is equal to the real part, and the second element to the imaginary part, of the complex number.
- 14 The type **char**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.³⁴⁾

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

32) See “future language directions” (6.11.1).

33) A specification for imaginary types is in informative annex G.

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

- 15 The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*. The implementation shall define **char** to have the same range, representation, and behavior as either **signed char** or **unsigned char**.³⁵⁾
- 16 An *enumeration* comprises a set of named integer constant values. Each distinct enumeration constitutes a different *enumerated type*.
- 17 The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integer types*. The integer and real floating types are collectively called *real types*.
- 18 Integer and floating types are collectively called *arithmetic types*. Each arithmetic type belongs to one *type domain*: the *real type domain* comprises the real types, the *complex type domain* comprises the complex types.
- 19 The **void** type comprises an empty set of values; it is an incomplete type that cannot be completed.
- 20 Any number of *derived types* can be constructed from the object, function, and incomplete types, as follows:
- An *array type* describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*.³⁶⁾ Array types are characterized by their element type and by the number of elements in the array. An array type is said to be derived from its element type, and if its element type is *T*, the array type is sometimes called “array of *T*”. The construction of an array type from an element type is called “array type derivation”.
 - A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.
 - A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.
 - A *function type* describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called “function returning *T*”. The construction of a function type from a return type is called “function type derivation”.

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

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

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

These methods of constructing derived types can be applied recursively.

- 21 Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.³⁷⁾
- 22 An array type of unknown size is an incomplete type. It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). A structure or union type of unknown content (as described in 6.7.2.3) is an incomplete type. It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope.
- 23 Array, function, and pointer types are collectively called *derived declarator types*. A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.
- 24 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types.
- 25 Any type so far mentioned is an *unqualified type*. Each unqualified type has several *qualified versions* of its type,³⁸⁾ corresponding to the combinations of one, two, or all three of the **const**, **volatile**, and **restrict** qualifiers. The qualified or unqualified versions of a type are distinct types that belong to the same type category and have the same representation and alignment requirements.³⁹⁾ A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.
- 26 A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type.³⁹⁾ Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements. All pointers to structure types shall have the same representation and alignment requirements as each other. All pointers to union types shall have the same representation and alignment requirements as each other. Pointers to other types need not have the same

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

38) See 6.7.3 regarding qualified array and function types.

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