

## 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 <b>&lt;assert.h&gt;</b>	169
7.2.1	Program diagnostics	169
7.3	Complex arithmetic <b>&lt;complex.h&gt;</b>	170
7.3.1	Introduction	170
7.3.2	Conventions	170
7.3.3	Branch cuts	171
7.3.4	The <b>CX_LIMITED_RANGE</b> 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 <b>&lt;ctype.h&gt;</b>	181
7.4.1	Character classification functions	181
7.4.2	Character case mapping functions	184
7.5	Errors <b>&lt;errno.h&gt;</b>	186
7.6	Floating-point environment <b>&lt;fenv.h&gt;</b>	187
7.6.1	The <b>FENV_ACCESS</b> pragma	189
7.6.2	Floating-point exceptions	190
7.6.3	Rounding	193
7.6.4	Environment	194
7.7	Characteristics of floating types <b>&lt;float.h&gt;</b>	197
7.8	Format conversion of integer types <b>&lt;inttypes.h&gt;</b>	198
7.8.1	Macros for format specifiers	198
7.8.2	Functions for greatest-width integer types	199

7.9	Alternative spellings <code>&lt;iso646.h&gt;</code>	202
7.10	Sizes of integer types <code>&lt;limits.h&gt;</code>	203
7.11	Localization <code>&lt;locale.h&gt;</code>	204
7.11.1	Locale control	205
7.11.2	Numeric formatting convention inquiry	206
7.12	Mathematics <code>&lt;math.h&gt;</code>	212
7.12.1	Treatment of error conditions	214
7.12.2	The <code>FP_CONTRACT</code> pragma	215
7.12.3	Classification macros	216
7.12.4	Trigonometric functions	218
7.12.5	Hyperbolic functions	221
7.12.6	Exponential and logarithmic functions	223
7.12.7	Power and absolute-value functions	228
7.12.8	Error and gamma functions	230
7.12.9	Nearest integer functions	231
7.12.10	Remainder functions	235
7.12.11	Manipulation functions	236
7.12.12	Maximum, minimum, and positive difference functions	238
7.12.13	Floating multiply-add	239
7.12.14	Comparison macros	240
7.13	Nonlocal jumps <code>&lt;set jmp.h&gt;</code>	243
7.13.1	Save calling environment	243
7.13.2	Restore calling environment	244
7.14	Signal handling <code>&lt;signal.h&gt;</code>	246
7.14.1	Specify signal handling	247
7.14.2	Send signal	248
7.15	Variable arguments <code>&lt;stdarg.h&gt;</code>	249
7.15.1	Variable argument list access macros	249
7.16	Boolean type and values <code>&lt;stdbool.h&gt;</code>	253
7.17	Common definitions <code>&lt;stddef.h&gt;</code>	254
7.18	Integer types <code>&lt;stdint.h&gt;</code>	255
7.18.1	Integer types	255
7.18.2	Limits of specified-width integer types	257
7.18.3	Limits of other integer types	259
7.18.4	Macros for integer constants	260
7.19	Input/output <code>&lt;stdio.h&gt;</code>	262
7.19.1	Introduction	262
7.19.2	Streams	264
7.19.3	Files	266
7.19.4	Operations on files	268
7.19.5	File access functions	270
7.19.6	Formatted input/output functions	274
7.19.7	Character input/output functions	296
7.19.8	Direct input/output functions	301

7.19.9	File positioning functions	302
7.19.10	Error-handling functions	304
7.20	General utilities <b>&lt;stdlib.h&gt;</b>	306
7.20.1	Numeric conversion functions	307
7.20.2	Pseudo-random sequence generation functions	312
7.20.3	Memory management functions	313
7.20.4	Communication with the environment	315
7.20.5	Searching and sorting utilities	318
7.20.6	Integer arithmetic functions	320
7.20.7	Multibyte/wide character conversion functions	321
7.20.8	Multibyte/wide string conversion functions	323
7.21	String handling <b>&lt;string.h&gt;</b>	325
7.21.1	String function conventions	325
7.21.2	Copying functions	325
7.21.3	Concatenation functions	327
7.21.4	Comparison functions	328
7.21.5	Search functions	330
7.21.6	Miscellaneous functions	333
7.22	Type-generic math <b>&lt;tgmath.h&gt;</b>	335
7.23	Date and time <b>&lt;time.h&gt;</b>	338
7.23.1	Components of time	338
7.23.2	Time manipulation functions	339
7.23.3	Time conversion functions	341
7.24	Extended multibyte and wide character utilities <b>&lt;wchar.h&gt;</b>	348
7.24.1	Introduction	348
7.24.2	Formatted wide character input/output functions	349
7.24.3	Wide character input/output functions	367
7.24.4	General wide string utilities	371
7.24.5	Wide character time conversion functions	385
7.24.6	Extended multibyte/wide character conversion utilities	386
7.25	Wide character classification and mapping utilities <b>&lt;wctype.h&gt;</b>	393
7.25.1	Introduction	393
7.25.2	Wide character classification utilities	394
7.25.3	Wide character case mapping utilities	399
7.26	Future library directions	401
7.26.1	Complex arithmetic <b>&lt;complex.h&gt;</b>	401
7.26.2	Character handling <b>&lt;ctype.h&gt;</b>	401
7.26.3	Errors <b>&lt;errno.h&gt;</b>	401
7.26.4	Format conversion of integer types <b>&lt;inttypes.h&gt;</b>	401
7.26.5	Localization <b>&lt;locale.h&gt;</b>	401
7.26.6	Signal handling <b>&lt;signal.h&gt;</b>	401
7.26.7	Boolean type and values <b>&lt;stdbool.h&gt;</b>	401
7.26.8	Integer types <b>&lt;stdint.h&gt;</b>	401
7.26.9	Input/output <b>&lt;stdio.h&gt;</b>	402

7.26.10	General utilities <code>&lt;stdlib.h&gt;</code>	402
7.26.11	String handling <code>&lt;string.h&gt;</code>	402
7.26.12	Extended multibyte and wide character utilities	
	<code>&lt;wchar.h&gt;</code>	402
7.26.13	Wide character classification and mapping utilities	
	<code>&lt;wctype.h&gt;</code>	402
Annex A	(informative) Language syntax summary	403
A.1	Lexical grammar	403
A.2	Phrase structure grammar	409
A.3	Preprocessing directives	416
Annex B	(informative) Library summary	418
B.1	Diagnostics <code>&lt;assert.h&gt;</code>	418
B.2	Complex <code>&lt;complex.h&gt;</code>	418
B.3	Character handling <code>&lt;ctype.h&gt;</code>	420
B.4	Errors <code>&lt;errno.h&gt;</code>	420
B.5	Floating-point environment <code>&lt;fenv.h&gt;</code>	420
B.6	Characteristics of floating types <code>&lt;float.h&gt;</code>	421
B.7	Format conversion of integer types <code>&lt;inttypes.h&gt;</code>	421
B.8	Alternative spellings <code>&lt;iso646.h&gt;</code>	422
B.9	Sizes of integer types <code>&lt;limits.h&gt;</code>	422
B.10	Localization <code>&lt;locale.h&gt;</code>	422
B.11	Mathematics <code>&lt;math.h&gt;</code>	422
B.12	Nonlocal jumps <code>&lt;setjmp.h&gt;</code>	427
B.13	Signal handling <code>&lt;signal.h&gt;</code>	427
B.14	Variable arguments <code>&lt;stdarg.h&gt;</code>	427
B.15	Boolean type and values <code>&lt;stdbool.h&gt;</code>	427
B.16	Common definitions <code>&lt;stddef.h&gt;</code>	428
B.17	Integer types <code>&lt;stdint.h&gt;</code>	428
B.18	Input/output <code>&lt;stdio.h&gt;</code>	428
B.19	General utilities <code>&lt;stdlib.h&gt;</code>	430
B.20	String handling <code>&lt;string.h&gt;</code>	432
B.21	Type-generic math <code>&lt;tgmath.h&gt;</code>	433
B.22	Date and time <code>&lt;time.h&gt;</code>	433
B.23	Extended multibyte/wide character utilities <code>&lt;wchar.h&gt;</code>	434
B.24	Wide character classification and mapping utilities <code>&lt;wctype.h&gt;</code>	436
Annex C	(informative) Sequence points	438
Annex D	(normative) Universal character names for identifiers	439
Annex E	(informative) Implementation limits	441
Annex F	(normative) IEC 60559 floating-point arithmetic	443
F.1	Introduction	443
F.2	Types	443
F.3	Operators and functions	444

F.4	Floating to integer conversion	446
F.5	Binary-decimal conversion	446
F.6	Contracted expressions	447
F.7	Floating-point environment	447
F.8	Optimization	450
F.9	Mathematics <code>&lt;math.h&gt;</code>	453
Annex G (informative)	IEC 60559-compatible complex arithmetic	466
G.1	Introduction	466
G.2	Types	466
G.3	Conventions	466
G.4	Conversions	467
G.5	Binary operators	467
G.6	Complex arithmetic <code>&lt;complex.h&gt;</code>	471
G.7	Type-generic math <code>&lt;tgmath.h&gt;</code>	479
Annex H (informative)	Language independent arithmetic	480
H.1	Introduction	480
H.2	Types	480
H.3	Notification	484
Annex I (informative)	Common warnings	486
Annex J (informative)	Portability issues	488
J.1	Unspecified behavior	488
J.2	Undefined behavior	491
J.3	Implementation-defined behavior	504
J.4	Locale-specific behavior	511
J.5	Common extensions	512
Bibliography		515
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.open-std.org/JTC1/SC22/WG14/> containing additional information relevant to this standard such as a Rationale for many of the decisions made during its preparation and a log of Defect Reports and Responses.
- 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>`
- treatment of error conditions by math library functions (`math_errhandling`)
- floating-point environment access in `<fenv.h>`
- IEC 60559 (also known as IEC 559 or IEEE arithmetic) support
- trailing comma allowed in `enum` declaration
- `%lf` conversion specifier allowed in `printf`
- inline functions
- the `snprintf` family of functions in `<stdio.h>`
- boolean type in `<stdbool.h>`
- idempotent type qualifiers
- empty macro arguments

- new structure type compatibility rules (tag compatibility)
  - additional predefined macro names
  - **\_Pragma** preprocessing operator
  - standard pragmas
  - **\_\_func\_\_** predefined identifier
  - **va\_copy** macro
  - additional **strftime** conversion specifiers
  - LIA compatibility annex
  - deprecate **ungetc** at the beginning of a binary file
  - remove deprecation of aliased array parameters
  - conversion of array to pointer not limited to lvalues
  - relaxed constraints on aggregate and union initialization
  - relaxed restrictions on portable header names
  - **return** without expression not permitted in function that returns a value (and vice versa)
- 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.<sup>1)</sup> It specifies
  - the representation of C programs;
  - the syntax and constraints of the C language;
  - the semantic rules for interpreting C programs;
  - the representation of input data to be processed by C programs;
  - the representation of output data produced by C programs;
  - the restrictions and limits imposed by a conforming implementation of C.
- 2 This International Standard does not specify
  - the mechanism by which C programs are transformed for use by a data-processing system;
  - the mechanism by which C programs are invoked for use by a data-processing system;
  - the mechanism by which input data are transformed for use by a C program;
  - the mechanism by which output data are transformed after being produced by a C program;
  - the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;

---

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.

— 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**

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

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

### 3.5

#### 1 **bit**

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

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

### 3.6

#### 1 **byte**

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

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

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

### 3.7

#### 1 **character**

<abstract> member of a set of elements used for the organization, control, or representation of data

#### 3.7.1

#### 1 **character**

single-byte character

<C> bit representation that fits in a byte

### 3.7.2

#### 1 **multibyte character**

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

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

### 3.7.3

#### 1 **wide character**

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.<sup>2)</sup> 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.<sup>3)</sup>

---

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.<sup>4)</sup>
- 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.<sup>5)</sup>
  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 tokens<sup>6)</sup> 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.<sup>7)</sup>
  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.<sup>8)</sup>

- 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;<sup>9)</sup> 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;<sup>10)</sup> 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*,<sup>11)</sup> 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) expression;
```

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

- 1 Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written (the *source character set*), and the set interpreted in the execution environment (the *execution character set*). Each set is further divided into a *basic character set*, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called *extended characters*. The combined set is also called the *extended character set*. The values of the members of the execution character set are implementation-defined.
- 2 In a character constant or string literal, members of the execution character set shall be represented by corresponding members of the source character set or by *escape sequences* consisting of the backslash `\` followed by one or more characters. A byte with all bits set to 0, called the *null character*, shall exist in the basic execution character set; it is used to terminate a character string.
- 3 Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the Latin alphabet

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

the 26 *lowercase letters* of the Latin alphabet

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

the 10 decimal *digits*

<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

the following 29 graphic characters

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

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*<sup>12)</sup>) 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. Such a byte shall not occur as part of any other multibyte character.

2 For source files, the following shall hold:

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

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

### 5.2.2 Character display semantics

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

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

`\a` (*alert*) Produces an audible or visible alert without changing the active position.

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

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

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

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

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

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

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

International Standard.

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

### 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:<sup>13)</sup>
  - 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 number of characters as the corresponding universal character name, if any)<sup>14)</sup>

---

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

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

- 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.

- 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`.<sup>15)</sup> 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.<sup>16)</sup> The following parameters are used to define the model for each floating-point type:

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

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

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

- 3 In addition to normalized floating-point numbers ( $f_1 > 0$  if  $x \neq 0$ ), floating types may be able to contain other kinds of floating-point numbers, such as subnormal floating-point numbers ( $x \neq 0$ ,  $e = e_{\min}$ ,  $f_1 = 0$ ) and unnormalized floating-point numbers ( $x \neq 0$ ,  $e > e_{\min}$ ,  $f_1 = 0$ ), and values that are not floating-point numbers, such as infinities and NaNs. A *NaN* is an encoding signifying Not-a-Number. A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception; a *signaling NaN* generally raises a floating-point exception when occurring as an arithmetic operand.<sup>17)</sup>
- 4 An implementation may give zero and non-numeric values (such as infinities and NaNs) a sign or may leave them unsigned. Wherever such values are unsigned, any requirement in this International Standard to retrieve the sign shall produce an unspecified sign, and any requirement to set the sign shall be ignored.

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.

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.

- 5 The accuracy of the floating-point operations (+, -, \*, /) and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results is implementation-defined, as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>`. The implementation may state that the accuracy is unknown.
- 6 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`.
- 7 The rounding mode for floating-point addition is characterized by the implementation-defined value of `FLT_ROUNDS`:<sup>18)</sup>
- 1 indeterminable
  - 0 toward zero
  - 1 to nearest
  - 2 toward positive infinity
  - 3 toward negative infinity

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

- 8 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`:<sup>19)</sup>
- 1 indeterminable;
  - 0 evaluate all operations and constants just to the range and precision of the type;

---

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.

- 9 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,

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

**FLT\_DIG** 6

**DBL\_DIG** 10

**LDBL\_DIG** 10



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

<b>FLT_MIN</b>	<b>1E-37</b>
<b>DBL_MIN</b>	<b>1E-37</b>
<b>LDBL_MIN</b>	<b>1E-37</b>

### Recommended practice

- 12 Conversion from (at least) **double** to decimal with **DECIMAL\_DIG** digits and back should be the identity function.
- 13 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**:

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

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

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

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

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

<b>FLT_RADIX</b>	<b>2</b>
<b>DECIMAL_DIG</b>	<b>17</b>
<b>FLT_MANT_DIG</b>	<b>24</b>
<b>FLT_EPSILON</b>	<b>1.19209290E-07F</b> // decimal constant
<b>FLT_EPSILON</b>	<b>0X1P-23F</b> // hex constant
<b>FLT_DIG</b>	<b>6</b>
<b>FLT_MIN_EXP</b>	<b>-125</b>
<b>FLT_MIN</b>	<b>1.17549435E-38F</b> // decimal constant
<b>FLT_MIN</b>	<b>0X1P-126F</b> // 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), extended multibyte and wide character utilities `<wchar.h>` (7.24), floating-point environment `<fenv.h>` (7.6), general utilities `<stdlib.h>` (7.20), input/output `<stdio.h>` (7.19), 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*.<sup>21)</sup> 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.<sup>22)</sup>
- 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,<sup>23)</sup> if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage.

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

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

### 6.2.3 Name spaces of identifiers

- 1 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. Thus, there are separate *name spaces* for various categories of identifiers, as follows:
  - *label names* (disambiguated by the syntax of the label declaration and use);
  - the *tags* of structures, unions, and enumerations (disambiguated by following any<sup>24)</sup> 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,<sup>25)</sup> and retains its last-stored value throughout its lifetime.<sup>26)</sup> 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.<sup>27)</sup> 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 nonnegative. If any other character is stored in a **char** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type.
- 4 There are five *standard signed integer types*, designated as **signed char**, **short int**, **int**, **long int**, and **long long int**. (These and other types may be designated in several additional ways, as described in 6.7.2.) There may also be implementation-defined *extended signed integer types*.<sup>28)</sup> The standard and extended signed integer types are collectively called *signed integer types*.<sup>29)</sup>
- 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*.<sup>30)</sup>

---

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.<sup>31)</sup> 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**.<sup>32)</sup> 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**.<sup>33)</sup> 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.<sup>34)</sup>

---

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**.<sup>35)</sup>
- 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*.<sup>36)</sup> 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*.<sup>37)</sup>
- 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,<sup>38)</sup> 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.<sup>39)</sup> 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.<sup>39)</sup> 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.

representation or alignment requirements.

- 27 EXAMPLE 1 The type designated as “**float \***” has type “pointer to **float**”. Its type category is pointer, not a floating type. The const-qualified version of this type is designated as “**float \* const**” whereas the type designated as “**const float \***” is not a qualified type — its type is “pointer to const-qualified **float**” and is a pointer to a qualified type.
- 28 EXAMPLE 2 The type designated as “**struct tag (\*[5])(float)**” has type “array of pointer to function returning **struct tag**”. The array has length five and the function has a single parameter of type **float**. Its type category is array.

**Forward references:** compatible type and composite type (6.2.7), declarations (6.7).

## 6.2.6 Representations of types

### 6.2.6.1 General

- 1 The representations of all types are unspecified except as stated in this subclause.
- 2 Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined.
- 3 Values stored in unsigned bit-fields and objects of type **unsigned char** shall be represented using a pure binary notation.<sup>40)</sup>
- 4 Values stored in non-bit-field objects of any other object type consist of  $n \times \text{CHAR\_BIT}$  bits, where  $n$  is the size of an object of that type, in bytes. The value may be copied into an object of type **unsigned char [n]** (e.g., by **memcpy**); the resulting set of bytes is called the *object representation* of the value. Values stored in bit-fields consist of  $m$  bits, where  $m$  is the size specified for the bit-field. The object representation is the set of  $m$  bits the bit-field comprises in the addressable storage unit holding it. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.
- 5 Certain object representations need not represent a value of the object type. If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.<sup>41)</sup> Such a representation is called a *trap representation*.

---

40) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems*.) A byte contains **CHAR\_BIT** bits, and the values of type **unsigned char** range from 0 to  $2^{\text{CHAR\_BIT}} - 1$ .

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

- 6 When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.<sup>42)</sup> The value of a structure or union object is never a trap representation, even though the value of a member of the structure or union object may be a trap representation.
- 7 When a value is stored in a member of an object of union type, the bytes of the object representation that do not correspond to that member but do correspond to other members take unspecified values.
- 8 Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.<sup>43)</sup> Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated.

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

### 6.2.6.2 Integer types

- 1 For unsigned integer types other than **unsigned char**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). If there are  $N$  value bits, each bit shall represent a different power of 2 between 1 and  $2^{N-1}$ , so that objects of that type shall be capable of representing values from 0 to  $2^N - 1$  using a pure binary representation; this shall be known as the value representation. The values of any padding bits are unspecified.<sup>44)</sup>
- 2 For signed integer types, the bits of the object representation shall be divided into three groups: value bits, padding bits, and the sign bit. There need not be any padding bits; there shall be exactly one sign bit. Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are  $M$  value bits in the signed type and  $N$  in the unsigned type, then  $M \leq N$ ). If the sign bit

---

42) Thus, for example, structure assignment need not copy any padding bits.

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

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

is zero, it shall not affect the resulting value. If the sign bit is one, the value shall be modified in one of the following ways:

- the corresponding value with sign bit 0 is negated (*sign and magnitude*);
- the sign bit has the value  $-(2^N)$  (*two's complement*);
- the sign bit has the value  $-(2^N - 1)$  (*ones' complement*).

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

- 3 If the implementation supports negative zeros, they shall be generated only by:
  - the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with arguments that produce such a value;
  - the `+`, `-`, `*`, `/`, and `%` operators where one argument is a negative zero and the result is zero;
  - compound assignment operators based on the above cases.

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

- 4 If the implementation does not support negative zeros, the behavior of the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with arguments that would produce such a value is undefined.
- 5 The values of any padding bits are unspecified.<sup>45)</sup> A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type.
- 6 The *precision* of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. The *width* of an integer type is the same but including any sign bit; thus for unsigned integer types the two values are the same, while for signed integer types the width is one greater than the precision.

---

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

## 6.2.7 Compatible type and composite type

- 1 Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described in 6.7.2 for type specifiers, in 6.7.3 for type qualifiers, and in 6.7.5 for declarators.<sup>46)</sup> Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: If one is declared with a tag, the other shall be declared with the same tag. If both are complete types, then the following additional requirements apply: there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types, and such that if one member of a corresponding pair is declared with a name, the other member is declared with the same name. For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have the same widths. For two enumerations, corresponding members shall have the same values.
- 2 All declarations that refer to the same object or function shall have compatible type; otherwise, the behavior is undefined.
- 3 A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:
  - If one type is an array of known constant size, the composite type is an array of that size; otherwise, if one type is a variable length array, the composite type is that type.
  - If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list.
  - If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.

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

- 4 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible,<sup>47)</sup> if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type.

---

46) Two types need not be identical to be compatible.

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

- 5 EXAMPLE Given the following two file scope declarations:

```
int f(int (*)(), double (*)[3]);  
int f(int (*)(char *), double (*)[1]);
```

The resulting composite type for the function is:

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

## 6.3 Conversions

- 1 Several operators convert operand values from one type to another automatically. This subclause specifies the result required from such an *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*). The list in 6.3.1.8 summarizes the conversions performed by most ordinary operators; it is supplemented as required by the discussion of each operator in 6.5.
- 2 Conversion of an operand value to a compatible type causes no change to the value or the representation.

**Forward references:** cast operators (6.5.4).

### 6.3.1 Arithmetic operands

#### 6.3.1.1 Boolean, characters, and integers

- 1 Every integer type has an *integer conversion rank* defined as follows:
  - No two signed integer types shall have the same rank, even if they have the same representation.
  - The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.
  - The rank of **long long int** shall be greater than the rank of **long int**, which shall be greater than the rank of **int**, which shall be greater than the rank of **short int**, which shall be greater than the rank of **signed char**.
  - The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.
  - The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.
  - The rank of **char** shall equal the rank of **signed char** and **unsigned char**.
  - The rank of **\_Bool** shall be less than the rank of all other standard integer types.
  - The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).
  - The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank.
  - For all integer types **T1**, **T2**, and **T3**, if **T1** has greater rank than **T2** and **T2** has greater rank than **T3**, then **T1** has greater rank than **T3**.
- 2 The following may be used in an expression wherever an **int** or **unsigned int** may be used:

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

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

If an **int** can represent all values of the original type, the value is converted to an **int**; otherwise, it is converted to an **unsigned int**. These are called the *integer promotions*.<sup>48)</sup> All other types are unchanged by the integer promotions.

- 3 The integer promotions preserve value including sign. As discussed earlier, whether a “plain” **char** is treated as signed is implementation-defined.

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

### 6.3.1.2 Boolean type

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

### 6.3.1.3 Signed and unsigned integers

- 1 When a value with integer type is converted to another integer type other than **\_Bool**, if the value can be represented by the new type, it is unchanged.
- 2 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.<sup>49)</sup>
- 3 Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

### 6.3.1.4 Real floating and integer

- 1 When a finite value of real floating type is converted to an integer type other than **\_Bool**, the fractional part is discarded (i.e., the value is truncated toward zero). If the value of the integral part cannot be represented by the integer type, the behavior is undefined.<sup>50)</sup>
- 2 When a value of integer type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented

---

48) The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary **+**, **-**, and **~** operators, and to both operands of the shift operators, as specified by their respective subclauses.

49) The rules describe arithmetic on the mathematical value, not the value of a given type of expression.

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

exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined.

### 6.3.1.5 Real floating types

- 1 When a **float** is promoted to **double** or **long double**, or a **double** is promoted to **long double**, its value is unchanged.
- 2 When a **double** is demoted to **float**, a **long double** is demoted to **double** or **float**, or a value being represented in greater precision and range than required by its semantic type (see 6.3.1.8) is explicitly converted to its semantic type, if the value being converted can be represented exactly in the new type, it is unchanged. If the value being converted is in the range of values that can be represented but cannot be represented exactly, the result is either the nearest higher or nearest lower representable value, chosen in an implementation-defined manner. If the value being converted is outside the range of values that can be represented, the behavior is undefined.

### 6.3.1.6 Complex types

- 1 When a value of complex type is converted to another complex type, both the real and imaginary parts follow the conversion rules for the corresponding real types.

### 6.3.1.7 Real and complex

- 1 When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero.
- 2 When a value of complex type is converted to a real type, the imaginary part of the complex value is discarded and the value of the real part is converted according to the conversion rules for the corresponding real type.

### 6.3.1.8 Usual arithmetic conversions

- 1 Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a *common real type* for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the *usual arithmetic conversions*:

First, if the corresponding real type of either operand is **long double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **long double**.

Otherwise, if the corresponding real type of either operand is **double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **double**.

Otherwise, if the corresponding real type of either operand is **float**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **float**.<sup>51)</sup>

Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:

If both operands have the same type, then no further conversion is needed.

Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.

Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.

Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.

Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

- 2 The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.<sup>52)</sup>

---

51) For example, addition of a **double** `_Complex` and a **float** entails just the conversion of the **float** operand to **double** (and yields a **double** `_Complex` result).

52) The cast and assignment operators are still required to perform their specified conversions as described in 6.3.1.4 and 6.3.1.5.

## 6.3.2 Other operands

### 6.3.2.1 Lvalues, arrays, and function designators

- 1 An *lvalue* is an expression with an object type or an incomplete type other than **void**;<sup>53)</sup> if an lvalue does not designate an object when it is evaluated, the behavior is undefined. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.
- 2 Except when it is the operand of the **sizeof** operator, the unary **&** operator, the **++** operator, the **--** operator, or the left operand of the **.** operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue). If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; otherwise, the value has the type of the lvalue. If the lvalue has an incomplete type and does not have array type, the behavior is undefined.
- 3 Except when it is the operand of the **sizeof** operator or the unary **&** operator, or is a string literal used to initialize an array, an expression that has type “array of *type*” is converted to an expression with type “pointer to *type*” that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.
- 4 A *function designator* is an expression that has function type. Except when it is the operand of the **sizeof** operator<sup>54)</sup> or the unary **&** operator, a function designator with type “function returning *type*” is converted to an expression that has type “pointer to function returning *type*”.

**Forward references:** address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions **<stddef.h>** (7.17), initialization (6.7.8), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** operator (6.5.3.4), structure and union members (6.5.2.3).

---

53) The name “lvalue” comes originally from the assignment expression **E1 = E2**, in which the left operand **E1** is required to be a (modifiable) lvalue. It is perhaps better considered as representing an object “locator value”. What is sometimes called “rvalue” is in this International Standard described as the “value of an expression”.

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

54) Because this conversion does not occur, the operand of the **sizeof** operator remains a function designator and violates the constraint in 6.5.3.4.

### 6.3.2.2 void

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

### 6.3.2.3 Pointers

- 1 A pointer to **void** may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to **void** and back again; the result shall compare equal to the original pointer.
- 2 For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; the values stored in the original and converted pointers shall compare equal.
- 3 An integer constant expression with the value 0, or such an expression cast to type **void \***, is called a *null pointer constant*.<sup>55)</sup> If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.
- 4 Conversion of a null pointer to another pointer type yields a null pointer of that type. Any two null pointers shall compare equal.
- 5 An integer may be converted to any pointer type. Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.<sup>56)</sup>
- 6 Any pointer type may be converted to an integer type. Except as previously specified, the result is implementation-defined. If the result cannot be represented in the integer type, the behavior is undefined. The result need not be in the range of values of any integer type.
- 7 A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned<sup>57)</sup> for the pointed-to type, the behavior is undefined. Otherwise, when converted back again, the result shall compare equal to the original pointer. When a pointer to an object is

---

55) The macro **NULL** is defined in `<stddef.h>` (and other headers) as a null pointer constant; see 7.17.

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

57) In general, the concept “correctly aligned” is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C.

converted to a pointer to a character type, the result points to the lowest addressed byte of the object. Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object.

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

**Forward references:** cast operators (6.5.4), equality operators (6.5.9), integer types capable of holding object pointers (7.18.1.4), simple assignment (6.5.16.1).

## 6.4 Lexical elements

### Syntax

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

### Constraints

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

### Semantics

- 3        A *token* is the minimal lexical element of the language in translation phases 7 and 8. The categories of tokens are: keywords, identifiers, constants, string literals, and punctuators. A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories.<sup>58)</sup> If a ' or a " character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by *white space*; this consists of comments (described later), or *white-space characters* (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in 6.10, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space may appear within a preprocessing token only as part of a header name or between the quotation characters in a character constant or string literal.

---

58) An additional category, placemarkers, is used internally in translation phase 4 (see 6.10.3.3); it cannot occur in source files.

- 4 If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token. There is one exception to this rule: a header name preprocessing token is only recognized within a **#include** preprocessing directive, and within such a directive, a sequence of characters that could be either a header name or a string literal is recognized as the former.
- 5 **EXAMPLE 1** The program fragment **1Ex** is parsed as a preprocessing number token (one that is not a valid floating or integer constant token), even though a parse as the pair of preprocessing tokens **1** and **Ex** might produce a valid expression (for example, if **Ex** were a macro defined as **+1**). Similarly, the program fragment **1E1** is parsed as a preprocessing number (one that is a valid floating constant token), whether or not **E** is a macro name.
- 6 **EXAMPLE 2** The program fragment **x+++++y** is parsed as **x ++ ++ + y**, which violates a constraint on increment operators, even though the parse **x ++ + ++ y** might yield a correct expression.

**Forward references:** character constants (6.4.4.4), comments (6.4.9), expressions (6.5), floating constants (6.4.4.2), header names (6.4.7), macro replacement (6.10.3), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), preprocessing directives (6.10), preprocessing numbers (6.4.8), string literals (6.4.5).

## 6.4.1 Keywords

### Syntax

- 1 *keyword:* one of
- |                 |                 |                 |                   |
|-----------------|-----------------|-----------------|-------------------|
| <b>auto</b>     | <b>enum</b>     | <b>restrict</b> | <b>unsigned</b>   |
| <b>break</b>    | <b>extern</b>   | <b>return</b>   | <b>void</b>       |
| <b>case</b>     | <b>float</b>    | <b>short</b>    | <b>volatile</b>   |
| <b>char</b>     | <b>for</b>      | <b>signed</b>   | <b>while</b>      |
| <b>const</b>    | <b>goto</b>     | <b>sizeof</b>   | <b>_Bool</b>      |
| <b>continue</b> | <b>if</b>       | <b>static</b>   | <b>_Complex</b>   |
| <b>default</b>  | <b>inline</b>   | <b>struct</b>   | <b>_Imaginary</b> |
| <b>do</b>       | <b>int</b>      | <b>switch</b>   |                   |
| <b>double</b>   | <b>long</b>     | <b>typedef</b>  |                   |
| <b>else</b>     | <b>register</b> | <b>union</b>    |                   |

### Semantics

- 2 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise. The keyword **\_Imaginary** is reserved for specifying imaginary types.<sup>59)</sup>

<sup>59)</sup> One possible specification for imaginary types appears in annex G.

## 6.4.2 Identifiers

### 6.4.2.1 General

#### Syntax

- 1        *identifier*:
- identifier-nondigit*  
*identifier identifier-nondigit*  
*identifier digit*
- identifier-nondigit*:
- nondigit*  
*universal-character-name*  
other implementation-defined characters
- nondigit*: one of
- |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| _ | a | b | c | d | e | f | g | h | i | j | k | l | m |
|   | n | o | p | q | r | s | t | u | v | w | x | y | z |
|   | A | B | C | D | E | F | G | H | I | J | K | L | M |
|   | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
- digit*: one of
- |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

#### Semantics

- 2        An identifier is a sequence of nondigit characters (including the underscore `_`, the lowercase and uppercase Latin letters, and other characters) and digits, which designates one or more entities as described in 6.2.1. Lowercase and uppercase letters are distinct. There is no specific limit on the maximum length of an identifier.
- 3        Each universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in annex D.<sup>60)</sup> The initial character shall not be a universal character name designating a digit. An implementation may allow multibyte characters that are not part of the basic source character set to appear in identifiers; which characters and their correspondence to universal character names is implementation-defined.
- 4        When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword.

---

60) On systems in which linkers cannot accept extended characters, an encoding of the universal character name may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the `\u` in a universal character name. Extended characters may produce a long external identifier.

### Implementation limits

- 5 As discussed in 5.2.4.1, an implementation may limit the number of significant initial characters in an identifier; the limit for an *external name* (an identifier that has external linkage) may be more restrictive than that for an *internal name* (a macro name or an identifier that does not have external linkage). The number of significant characters in an identifier is implementation-defined.
- 6 Any identifiers that differ in a significant character are different identifiers. If two identifiers differ only in nonsignificant characters, the behavior is undefined.

**Forward references:** universal character names (6.4.3), macro replacement (6.10.3).

### 6.4.2.2 Predefined identifiers

#### Semantics

- 1 The identifier `__func__` shall be implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration

```
static const char __func__[ ] = "function-name";
```

appeared, where *function-name* is the name of the lexically-enclosing function.<sup>61)</sup>

- 2 This name is encoded as if the implicit declaration had been written in the source character set and then translated into the execution character set as indicated in translation phase 5.
- 3 **EXAMPLE** Consider the code fragment:

```
#include <stdio.h>
void myfunc(void)
{
    printf("%s\n", __func__);
    /* ... */
}
```

Each time the function is called, it will print to the standard output stream:

```
myfunc
```

**Forward references:** function definitions (6.9.1).

---

61) Since the name `__func__` is reserved for any use by the implementation (7.1.3), if any other identifier is explicitly declared using the name `__func__`, the behavior is undefined.

### 6.4.3 Universal character names

#### Syntax

- 1        *universal-character-name*:  
           u *hex-quad*  
           U *hex-quad hex-quad*
- hex-quad*:  
           *hexadecimal-digit hexadecimal-digit*  
           *hexadecimal-digit hexadecimal-digit*

#### Constraints

- 2        A universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024 (\$), 0040 (@), or 0060 (´), nor one in the range D800 through DFFF inclusive.<sup>62)</sup>

#### Description

- 3        Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set.

#### Semantics

- 4        The universal character name `\Uxxxxxxxx` designates the character whose eight-digit short identifier (as specified by ISO/IEC 10646) is *xxxxxxxx*.<sup>63)</sup> Similarly, the universal character name `\uxxxx` designates the character whose four-digit short identifier is *xxxx* (and whose eight-digit short identifier is *0000xxxx*).

---

62) The disallowed characters are the characters in the basic character set and the code positions reserved by ISO/IEC 10646 for control characters, the character DELETE, and the S-zone (reserved for use by UTF-16).

63) Short identifiers for characters were first specified in ISO/IEC 10646-1/AMD9:1997.

## 6.4.4 Constants

### Syntax

- 1        *constant*:
- integer-constant*
  - floating-constant*
  - enumeration-constant*
  - character-constant*

### Constraints

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

### Semantics

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

#### 6.4.4.1 Integer constants

### Syntax

- 1        *integer-constant*:
- decimal-constant integer-suffix<sub>opt</sub>*
  - octal-constant integer-suffix<sub>opt</sub>*
  - hexadecimal-constant integer-suffix<sub>opt</sub>*
- decimal-constant*:
- nonzero-digit*
  - decimal-constant digit*
- octal-constant*:
- 0**
  - octal-constant octal-digit*
- hexadecimal-constant*:
- hexadecimal-prefix hexadecimal-digit*
  - hexadecimal-constant hexadecimal-digit*
- hexadecimal-prefix*: one of
- 0x 0X**
- nonzero-digit*: one of
- 1 2 3 4 5 6 7 8 9**
- octal-digit*: one of
- 0 1 2 3 4 5 6 7**

*hexadecimal-digit*: one of

**0 1 2 3 4 5 6 7 8 9**  
**a b c d e f**  
**A B C D E F**

*integer-suffix*:

*unsigned-suffix long-suffix<sub>opt</sub>*  
*unsigned-suffix long-long-suffix*  
*long-suffix unsigned-suffix<sub>opt</sub>*  
*long-long-suffix unsigned-suffix<sub>opt</sub>*

*unsigned-suffix*: one of

**u U**

*long-suffix*: one of

**l L**

*long-long-suffix*: one of

**ll LL**

### Description

- 2 An integer constant begins with a digit, but has no period or exponent part. It may have a prefix that specifies its base and a suffix that specifies its type.
- 3 A decimal constant begins with a nonzero digit and consists of a sequence of decimal digits. An octal constant consists of the prefix **0** optionally followed by a sequence of the digits **0** through **7** only. A hexadecimal constant consists of the prefix **0x** or **0X** followed by a sequence of the decimal digits and the letters **a** (or **A**) through **f** (or **F**) with values 10 through 15 respectively.

### Semantics

- 4 The value of a decimal constant is computed base 10; that of an octal constant, base 8; that of a hexadecimal constant, base 16. The lexically first digit is the most significant.
- 5 The type of an integer constant is the first of the corresponding list in which its value can be represented.

Suffix	Decimal Constant	Octal or Hexadecimal Constant
none	int long int long long int	int unsigned int long int unsigned long int long long int unsigned long long int
u or U	unsigned int unsigned long int unsigned long long int	unsigned int unsigned long int unsigned long long int
l or L	long int long long int	long int unsigned long int long long int unsigned long long int
Both u or U and l or L	unsigned long int unsigned long long int	unsigned long int unsigned long long int
ll or LL	long long int	long long int unsigned long long int
Both u or U and ll or LL	unsigned long long int	unsigned long long int

If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. If all of the types in the list for the constant are signed, the extended integer type shall be signed. If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned.

### 6.4.4.2 Floating constants

#### Syntax

- 1 *floating-constant*:
- decimal-floating-constant*  
*hexadecimal-floating-constant*
- decimal-floating-constant*:
- fractional-constant* *exponent-part*<sub>opt</sub> *floating-suffix*<sub>opt</sub>  
*digit-sequence* *exponent-part* *floating-suffix*<sub>opt</sub>
- hexadecimal-floating-constant*:
- hexadecimal-prefix* *hexadecimal-fractional-constant*  
*binary-exponent-part* *floating-suffix*<sub>opt</sub>  
*hexadecimal-prefix* *hexadecimal-digit-sequence*  
*binary-exponent-part* *floating-suffix*<sub>opt</sub>
- fractional-constant*:
- digit-sequence*<sub>opt</sub> . *digit-sequence*  
*digit-sequence* .
- exponent-part*:
- e** *sign*<sub>opt</sub> *digit-sequence*  
**E** *sign*<sub>opt</sub> *digit-sequence*
- sign*: one of
- + -
- digit-sequence*:
- digit*  
*digit-sequence* *digit*
- hexadecimal-fractional-constant*:
- hexadecimal-digit-sequence*<sub>opt</sub> .  
*hexadecimal-digit-sequence*  
*hexadecimal-digit-sequence* .
- binary-exponent-part*:
- p** *sign*<sub>opt</sub> *digit-sequence*  
**P** *sign*<sub>opt</sub> *digit-sequence*
- hexadecimal-digit-sequence*:
- hexadecimal-digit*  
*hexadecimal-digit-sequence* *hexadecimal-digit*
- floating-suffix*: one of
- f l F L**

### Description

- 2 A floating constant has a *significant part* that may be followed by an *exponent part* and a suffix that specifies its type. The components of the significant part may include a digit sequence representing the whole-number part, followed by a period (`.`), followed by a digit sequence representing the fraction part. The components of the exponent part are an `e`, `E`, `p`, or `P` followed by an exponent consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part has to be present; for decimal floating constants, either the period or the exponent part has to be present.

### Semantics

- 3 The significant part is interpreted as a (decimal or hexadecimal) rational number; the digit sequence in the exponent part is interpreted as a decimal integer. For decimal floating constants, the exponent indicates the power of 10 by which the significant part is to be scaled. For hexadecimal floating constants, the exponent indicates the power of 2 by which the significant part is to be scaled. For decimal floating constants, and also for hexadecimal floating constants when `FLT_RADIX` is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. For hexadecimal floating constants when `FLT_RADIX` is a power of 2, the result is correctly rounded.
- 4 An unsuffixed floating constant has type `double`. If suffixed by the letter `f` or `F`, it has type `float`. If suffixed by the letter `l` or `L`, it has type `long double`.
- 5 Floating constants are converted to internal format as if at translation-time. The conversion of a floating constant shall not raise an exceptional condition or a floating-point exception at execution time.

### Recommended practice

- 6 The implementation should produce a diagnostic message if a hexadecimal constant cannot be represented exactly in its evaluation format; the implementation should then proceed with the translation of the program.
- 7 The translation-time conversion of floating constants should match the execution-time conversion of character strings by library functions, such as `strtod`, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding.<sup>64)</sup>

---

64) The specification for the library functions recommends more accurate conversion than required for floating constants (see 7.20.1.3).

### 6.4.4.3 Enumeration constants

#### Syntax

1 *enumeration-constant:*  
*identifier*

#### Semantics

2 An identifier declared as an enumeration constant has type **int**.

**Forward references:** enumeration specifiers (6.7.2.2).

### 6.4.4.4 Character constants

#### Syntax

1 *character-constant:*  
*' c-char-sequence '*  
*␣' c-char-sequence ␣*

*c-char-sequence:*  
*c-char*  
*c-char-sequence c-char*

*c-char:*  
any member of the source character set except  
the single-quote **'**, backslash **\**, or new-line character  
*escape-sequence*

*escape-sequence:*  
*simple-escape-sequence*  
*octal-escape-sequence*  
*hexadecimal-escape-sequence*  
*universal-character-name*

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

*octal-escape-sequence:*  
**\ octal-digit**  
**\ octal-digit octal-digit**  
**\ octal-digit octal-digit octal-digit**

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

## Description

- 2 An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in `'x'`. A wide character constant is the same, except prefixed by the letter `L`. With a few exceptions detailed later, the elements of the sequence are any members of the source character set; they are mapped in an implementation-defined manner to members of the execution character set.
- 3 The single-quote `'`, the double-quote `"`, the question-mark `?`, the backslash `\`, and arbitrary integer values are representable according to the following table of escape sequences:
- |                              |                                   |
|------------------------------|-----------------------------------|
| single quote <code>'</code>  | <code>\'</code>                   |
| double quote <code>"</code>  | <code>\"</code>                   |
| question mark <code>?</code> | <code>\?</code>                   |
| backslash <code>\</code>     | <code>\\</code>                   |
| octal character              | <code>\octal digits</code>        |
| hexadecimal character        | <code>\xhexadecimal digits</code> |
- 4 The double-quote `"` and question-mark `?` are representable either by themselves or by the escape sequences `\"` and `\?`, respectively, but the single-quote `'` and the backslash `\` shall be represented, respectively, by the escape sequences `\'` and `\\`.
- 5 The octal digits that follow the backslash in an octal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the octal integer so formed specifies the value of the desired character or wide character.
- 6 The hexadecimal digits that follow the backslash and the letter `x` in a hexadecimal escape sequence are taken to be part of the construction of a single character for an integer character constant or of a single wide character for a wide character constant. The numerical value of the hexadecimal integer so formed specifies the value of the desired character or wide character.
- 7 Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.
- 8 In addition, characters not in the basic character set are representable by universal character names and certain nongraphic characters are representable by escape sequences consisting of the backslash `\` followed by a lowercase letter: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`.<sup>65)</sup>

---

65) The semantics of these characters were discussed in 5.2.2. If any other character follows a backslash, the result is not a token and a diagnostic is required. See “future language directions” (6.11.4).

### Constraints

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

### Semantics

- 10 An integer character constant has type **int**. The value of an integer character constant containing a single character that maps to a single-byte execution character is the numerical value of the representation of the mapped character interpreted as an integer. The value of an integer character constant containing more than one character (e.g., **'ab'**), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined. If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **char** whose value is that of the single character or escape sequence is converted to type **int**.
- 11 A wide character constant has type **wchar\_t**, an integer type defined in the **<stddef.h>** header. The value of a wide character constant containing a single multibyte character that maps to a member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the **mbtowc** function, with an implementation-defined current locale. The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined.
- 12 EXAMPLE 1 The construction **'\0'** is commonly used to represent the null character.
- 13 EXAMPLE 2 Consider implementations that use two's-complement representation for integers and eight bits for objects that have type **char**. In an implementation in which type **char** has the same range of values as **signed char**, the integer character constant **'\xFF'** has the value **-1**; if type **char** has the same range of values as **unsigned char**, the character constant **'\xFF'** has the value **+255**.
- 14 EXAMPLE 3 Even if eight bits are used for objects that have type **char**, the construction **'\x123'** specifies an integer character constant containing only one character, since a hexadecimal escape sequence is terminated only by a non-hexadecimal character. To specify an integer character constant containing the two characters whose values are **'\x12'** and **'3'**, the construction **'\0223'** may be used, since an octal escape sequence is terminated after three octal digits. (The value of this two-character integer character constant is implementation-defined.)
- 15 EXAMPLE 4 Even if 12 or more bits are used for objects that have type **wchar\_t**, the construction **L'\1234'** specifies the implementation-defined value that results from the combination of the values **0123** and **'4'**.

**Forward references:** common definitions **<stddef.h>** (7.17), the **mbtowc** function (7.20.7.2).

## 6.4.5 String literals

### Syntax

- 1 *string-literal*:
- ```
" s-char-sequenceopt "
```
- ```
L" s-char-sequenceopt "
```
- s-char-sequence*:
- ```
s-char
```
- ```
s-char-sequence s-char
```
- s-char*:
- any member of the source character set except  
the double-quote " , backslash \ , or new-line character  
*escape-sequence*

### Description

- 2 A *character string literal* is a sequence of zero or more multibyte characters enclosed in double-quotes, as in "**xyz**". A *wide string literal* is the same, except prefixed by the letter **L**.
- 3 The same considerations apply to each element of the sequence in a character string literal or a wide string literal as if it were in an integer character constant or a wide character constant, except that the single-quote ' is representable either by itself or by the escape sequence \ ', but the double-quote " shall be represented by the escape sequence \ " .

### Semantics

- 4 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and wide string literal tokens are concatenated into a single multibyte character sequence. If any of the tokens are wide string literal tokens, the resulting multibyte character sequence is treated as a wide string literal; otherwise, it is treated as a character string literal.
- 5 In translation phase 7, a byte or code of value zero is appended to each multibyte character sequence that results from a string literal or literals.<sup>66)</sup> The multibyte character sequence is then used to initialize an array of static storage duration and length just sufficient to contain the sequence. For character string literals, the array elements have type **char**, and are initialized with the individual bytes of the multibyte character sequence; for wide string literals, the array elements have type **wchar\_t**, and are initialized with the sequence of wide characters corresponding to the multibyte character

---

66) A character string literal need not be a string (see 7.1.1), because a null character may be embedded in it by a \0 escape sequence.

sequence, as defined by the `mbstowcs` function with an implementation-defined current locale. The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

- 6 It is unspecified whether these arrays are distinct provided their elements have the appropriate values. If the program attempts to modify such an array, the behavior is undefined.

- 7 EXAMPLE This pair of adjacent character string literals

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

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

**Forward references:** common definitions `<stddef.h>` (7.17), the `mbstowcs` function (7.20.8.1).

## 6.4.6 Punctuators

### Syntax

- 1 *punctuator*: one of
- ```
[ ] ( ) { } . ->
++ -- & * + - ~ !
/ % << >> < > <= >= == != ^ | && ||
? : ; ...
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
<: :> <% %> %: %:::
```

### Semantics

- 2 A punctuator is a symbol that has independent syntactic and semantic significance. Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an *operator* (other forms of operator also exist in some contexts). An *operand* is an entity on which an operator acts.

- 3 In all aspects of the language, the six tokens<sup>67)</sup>

`<:    :>    <%    %>    %:    %:::`

behave, respectively, the same as the six tokens

`[    ]    {    }    #    ##`

except for their spelling.<sup>68)</sup>

**Forward references:** expressions (6.5), declarations (6.7), preprocessing directives (6.10), statements (6.8).

### 6.4.7 Header names

#### Syntax

- 1 *header-name:*  
     `< h-char-sequence >`  
     `" q-char-sequence "`
- h-char-sequence:*  
     `h-char`  
     `h-char-sequence h-char`
- h-char:*  
     any member of the source character set except  
     the new-line character and `>`
- q-char-sequence:*  
     `q-char`  
     `q-char-sequence q-char`
- q-char:*  
     any member of the source character set except  
     the new-line character and `"`

#### Semantics

- 2 The sequences in both forms of header names are mapped in an implementation-defined manner to headers or external source file names as specified in 6.10.2.
- 3 If the characters `'`, `\`, `"`, `//`, or `/*` occur in the sequence between the `<` and `>` delimiters, the behavior is undefined. Similarly, if the characters `'`, `\`, `//`, or `/*` occur in the

---

67) These tokens are sometimes called “digraphs”.

68) Thus `[` and `<:` behave differently when “stringized” (see 6.10.3.2), but can otherwise be freely interchanged.

sequence between the " delimiters, the behavior is undefined.<sup>69)</sup> A header name preprocessing token is recognized only within a **#include** preprocessing directive.

- 4 EXAMPLE The following sequence of characters:

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

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

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

**Forward references:** source file inclusion (6.10.2).

## 6.4.8 Preprocessing numbers

### Syntax

- 1 *pp-number*:
- digit*
  - .* *digit*
  - pp-number digit*
  - pp-number identifier-nondigit*
  - pp-number e sign*
  - pp-number E sign*
  - pp-number p sign*
  - pp-number P sign*
  - pp-number .*

### Description

- 2 A preprocessing number begins with a digit optionally preceded by a period (.) and may be followed by valid identifier characters and the character sequences **e+**, **e-**, **E+**, **E-**, **p+**, **p-**, **P+**, or **P-**.
- 3 Preprocessing number tokens lexically include all floating and integer constant tokens.

### Semantics

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

---

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

## 6.4.9 Comments

- 1 Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it.<sup>70)</sup>
- 2 Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.
- 3 EXAMPLE

```

"a//b"                // four-character string literal
#include "//e"        // undefined behavior
// */                // comment, not syntax error
f = g/**//h;         // equivalent to f = g / h;
//\
i();                 // part of a two-line comment
/\
/ j();              // part of a two-line comment
#define glue(x,y) x##y
glue(/,/ ) k();      // syntax error; not comment
/**// l();          // equivalent to l();
m = n/**/o
  + p;              // equivalent to m = n + p;

```

---

70) Thus, `/* ... */` comments do not nest.

## 6.5 Expressions

- 1 An *expression* is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof.
- 2 Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.<sup>71)</sup>
- 3 The grouping of operators and operands is indicated by the syntax.<sup>72)</sup> Except as specified later (for the function-call ( ), &&, | |, ?:, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.
- 4 Some operators (the unary operator ~, and the binary operators <<, >>, &, ^, and |, collectively described as *bitwise operators*) are required to have operands that have integer type. These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types.
- 5 If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined.
- 6 The *effective type* of an object for an access to its stored value is the declared type of the object, if any.<sup>73)</sup> If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify

---

71) This paragraph renders undefined statement expressions such as

```
i = ++i + 1;
a[i++] = i;
```

while allowing

```
i = i + 1;
a[i] = i;
```

72) The syntax specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first. Thus, for example, the expressions allowed as the operands of the binary + operator (6.5.6) are those expressions defined in 6.5.1 through 6.5.6. The exceptions are cast expressions (6.5.4) as operands of unary operators (6.5.3), and an operand contained between any of the following pairs of operators: grouping parentheses ( ) (6.5.1), subscripting brackets [ ] (6.5.2.1), function-call parentheses ( ) (6.5.2.2), and the conditional operator ?: (6.5.15).

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

73) Allocated objects have no declared type.

the stored value. If a value is copied into an object having no declared type using **memcpy** or **memmove**, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.

- 7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:<sup>74)</sup>
- a type compatible with the effective type of the object,
  - a qualified version of a type compatible with the effective type of the object,
  - a type that is the signed or unsigned type corresponding to the effective type of the object,
  - a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
  - an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
  - a character type.
- 8 A floating expression may be *contracted*, that is, evaluated as though it were an atomic operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.<sup>75)</sup> The **FP\_CONTRACT** pragma in **<math.h>** provides a way to disallow contracted expressions. Otherwise, whether and how expressions are contracted is implementation-defined.<sup>76)</sup>

**Forward references:** the **FP\_CONTRACT** pragma (7.12.2), copying functions (7.21.2).

---

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

75) A contracted expression might also omit the raising of floating-point exceptions.

76) This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. As contractions potentially undermine predictability, and can even decrease accuracy for containing expressions, their use needs to be well-defined and clearly documented.

## 6.5.1 Primary expressions

### Syntax

- 1        *primary-expression:*  
           *identifier*  
           *constant*  
           *string-literal*  
           ( *expression* )

### Semantics

- 2        An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).<sup>77)</sup>
- 3        A constant is a primary expression. Its type depends on its form and value, as detailed in 6.4.4.
- 4        A string literal is a primary expression. It is an lvalue with type as detailed in 6.4.5.
- 5        A parenthesized expression is a primary expression. Its type and value are identical to those of the unparenthesized expression. It is an lvalue, a function designator, or a void expression if the unparenthesized expression is, respectively, an lvalue, a function designator, or a void expression.

**Forward references:** declarations (6.7).

## 6.5.2 Postfix operators

### Syntax

- 1        *postfix-expression:*  
           *primary-expression*  
           *postfix-expression* [ *expression* ]  
           *postfix-expression* ( *argument-expression-list*<sub>opt</sub> )  
           *postfix-expression* . *identifier*  
           *postfix-expression* -> *identifier*  
           *postfix-expression* ++  
           *postfix-expression* --  
           ( *type-name* ) { *initializer-list* }  
           ( *type-name* ) { *initializer-list* , }

---

<sup>77)</sup> Thus, an undeclared identifier is a violation of the syntax.

*argument-expression-list:*  
*assignment-expression*  
*argument-expression-list* , *assignment-expression*

### 6.5.2.1 Array subscripting

#### Constraints

- 1 One of the expressions shall have type “pointer to object *type*”, the other expression shall have integer type, and the result has type “*type*”.

#### Semantics

- 2 A postfix expression followed by an expression in square brackets [ ] is a subscripted designation of an element of an array object. The definition of the subscript operator [ ] is that **E1[E2]** is identical to **(\*( (E1)+(E2) ) )**. Because of the conversion rules that apply to the binary + operator, if **E1** is an array object (equivalently, a pointer to the initial element of an array object) and **E2** is an integer, **E1[E2]** designates the **E2**-th element of **E1** (counting from zero).
- 3 Successive subscript operators designate an element of a multidimensional array object. If **E** is an *n*-dimensional array ( $n \geq 2$ ) with dimensions  $i \times j \times \dots \times k$ , then **E** (used as other than an lvalue) is converted to a pointer to an  $(n - 1)$ -dimensional array with dimensions  $j \times \dots \times k$ . If the unary \* operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the pointed-to  $(n - 1)$ -dimensional array, which itself is converted into a pointer if used as other than an lvalue. It follows from this that arrays are stored in row-major order (last subscript varies fastest).
- 4 EXAMPLE Consider the array object defined by the declaration

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

Here **x** is a  $3 \times 5$  array of **ints**; more precisely, **x** is an array of three element objects, each of which is an array of five **ints**. In the expression **x[i]**, which is equivalent to **(\*( (x)+(i) ) )**, **x** is first converted to a pointer to the initial array of five **ints**. Then **i** is adjusted according to the type of **x**, which conceptually entails multiplying **i** by the size of the object to which the pointer points, namely an array of five **int** objects. The results are added and indirection is applied to yield an array of five **ints**. When used in the expression **x[i][j]**, that array is in turn converted to a pointer to the first of the **ints**, so **x[i][j]** yields an **int**.

**Forward references:** additive operators (6.5.6), address and indirection operators (6.5.3.2), array declarators (6.7.5.2).

### 6.5.2.2 Function calls

#### Constraints

- 1 The expression that denotes the called function<sup>78)</sup> shall have type pointer to function returning **void** or returning an object type other than an array type.
- 2 If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter.

#### Semantics

- 3 A postfix expression followed by parentheses ( ) containing a possibly empty, comma-separated list of expressions is a function call. The postfix expression denotes the called function. The list of expressions specifies the arguments to the function.
- 4 An argument may be an expression of any object type. In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.<sup>79)</sup>
- 5 If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4. Otherwise, the function call has type **void**. If an attempt is made to modify the result of a function call or to access it after the next sequence point, the behavior is undefined.
- 6 If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **float** are promoted to **double**. These are called the *default argument promotions*. If the number of arguments does not equal the number of parameters, the behavior is undefined. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (, ...) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined. If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases:

---

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

79) A function may change the values of its parameters, but these changes cannot affect the values of the arguments. On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. A parameter declared to have array or function type is adjusted to have a pointer type as described in 6.9.1.

- one promoted type is a signed integer type, the other promoted type is the corresponding unsigned integer type, and the value is representable in both types;
  - both types are pointers to qualified or unqualified versions of a character type or **void**.
- 7 If the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.
  - 8 No other conversions are performed implicitly; in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.
  - 9 If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined.
  - 10 The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call.
  - 11 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.
  - 12 **EXAMPLE** In the function call

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

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

**Forward references:** function declarators (including prototypes) (6.7.5.3), function definitions (6.9.1), the **return** statement (6.8.6.4), simple assignment (6.5.16.1).

### 6.5.2.3 Structure and union members

#### Constraints

- 1 The first operand of the **.** operator shall have a qualified or unqualified structure or union type, and the second operand shall name a member of that type.
- 2 The first operand of the **->** operator shall have type “pointer to qualified or unqualified structure” or “pointer to qualified or unqualified union”, and the second operand shall name a member of the type pointed to.

## Semantics

- 3 A postfix expression followed by the `.` operator and an identifier designates a member of a structure or union object. The value is that of the named member, and is an lvalue if the first expression is an lvalue. If the first expression has qualified type, the result has the so-qualified version of the type of the designated member.
- 4 A postfix expression followed by the `->` operator and an identifier designates a member of a structure or union object. The value is that of the named member of the object to which the first expression points, and is an lvalue.<sup>80)</sup> If the first expression is a pointer to a qualified type, the result has the so-qualified version of the type of the designated member.
- 5 One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the complete type of the union is visible. Two structures share a *common initial sequence* if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.
- 6 EXAMPLE 1 If `f` is a function returning a structure or union, and `x` is a member of that structure or union, `f().x` is a valid postfix expression but is not an lvalue.
- 7 EXAMPLE 2 In:

```

struct s { int i; const int ci; };
struct s s;
const struct s cs;
volatile struct s vs;

```

the various members have the types:

```

s.i      int
s.ci     const int
cs.i     const int
cs.ci    const int
vs.i     volatile int
vs.ci    volatile const int

```

---

80) If `&E` is a valid pointer expression (where `&` is the “address-of” operator, which generates a pointer to its operand), the expression `(&E)->MOS` is the same as `E.MOS`.

8 EXAMPLE 3 The following is a valid fragment:

```

union {
    struct {
        int    alltypes;
    } n;
    struct {
        int    type;
        int    intnode;
    } ni;
    struct {
        int    type;
        double doublenode;
    } nf;
} u;
u.nf.type = 1;
u.nf.doublenode = 3.14;
/* ... */
if (u.n.alltypes == 1)
    if (sin(u.nf.doublenode) == 0.0)
        /* ... */

```

The following is not a valid fragment (because the union type is not visible within function `f`):

```

struct t1 { int m; };
struct t2 { int m; };
int f(struct t1 *p1, struct t2 *p2)
{
    if (p1->m < 0)
        p2->m = -p2->m;
    return p1->m;
}
int g()
{
    union {
        struct t1 s1;
        struct t2 s2;
    } u;
    /* ... */
    return f(&u.s1, &u.s2);
}

```

**Forward references:** address and indirection operators (6.5.3.2), structure and union specifiers (6.7.2.1).

#### 6.5.2.4 Postfix increment and decrement operators

##### Constraints

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

##### Semantics

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

**Forward references:** additive operators (6.5.6), compound assignment (6.5.16.2).

#### 6.5.2.5 Compound literals

##### Constraints

- 1 The type name shall specify an object type or an array of unknown size, but not a variable length array type.
- 2 No initializer shall attempt to provide a value for an object not contained within the entire unnamed object specified by the compound literal.
- 3 If the compound literal occurs outside the body of a function, the initializer list shall consist of constant expressions.

##### Semantics

- 4 A postfix expression that consists of a parenthesized type name followed by a brace-enclosed list of initializers is a *compound literal*. It provides an unnamed object whose value is given by the initializer list.<sup>81)</sup>
- 5 If the type name specifies an array of unknown size, the size is determined by the initializer list as specified in 6.7.8, and the type of the compound literal is that of the completed array type. Otherwise (when the type name specifies an object type), the type of the compound literal is that specified by the type name. In either case, the result is an lvalue.

---

81) Note that this differs from a cast expression. For example, a cast specifies a conversion to scalar types or `void` only, and the result of a cast expression is not an lvalue.

- 6 The value of the compound literal is that of an unnamed object initialized by the initializer list. If the compound literal occurs outside the body of a function, the object has static storage duration; otherwise, it has automatic storage duration associated with the enclosing block.
- 7 All the semantic rules and constraints for initializer lists in 6.7.8 are applicable to compound literals.<sup>82)</sup>
- 8 String literals, and compound literals with const-qualified types, need not designate distinct objects.<sup>83)</sup>
- 9 EXAMPLE 1 The file scope definition

```
int *p = (int []){2, 4};
```

initializes **p** to point to the first element of an array of two ints, the first having the value two and the second, four. The expressions in this compound literal are required to be constant. The unnamed object has static storage duration.

- 10 EXAMPLE 2 In contrast, in

```
void f(void)
{
    int *p;
    /*...*/
    p = (int [2]){*p};
    /*...*/
}
```

**p** is assigned the address of the first element of an array of two ints, the first having the value previously pointed to by **p** and the second, zero. The expressions in this compound literal need not be constant. The unnamed object has automatic storage duration.

- 11 EXAMPLE 3 Initializers with designations can be combined with compound literals. Structure objects created using compound literals can be passed to functions without depending on member order:

```
drawline((struct point){.x=1, .y=1},
        (struct point){.x=3, .y=4});
```

Or, if **drawline** instead expected pointers to **struct point**:

```
drawline(&(struct point){.x=1, .y=1},
        &(struct point){.x=3, .y=4});
```

- 12 EXAMPLE 4 A read-only compound literal can be specified through constructions like:

```
(const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

---

82) For example, subobjects without explicit initializers are initialized to zero.

83) This allows implementations to share storage for string literals and constant compound literals with the same or overlapping representations.

- 13 EXAMPLE 5 The following three expressions have different meanings:

```
"/tmp/fileXXXXXX"
(char []){"/tmp/fileXXXXXX"}
(const char []){"/tmp/fileXXXXXX"}
```

The first always has static storage duration and has type array of `char`, but need not be modifiable; the last two have automatic storage duration when they occur within the body of a function, and the first of these two is modifiable.

- 14 EXAMPLE 6 Like string literals, `const`-qualified compound literals can be placed into read-only memory and can even be shared. For example,

```
(const char []){"abc"} == "abc"
```

might yield 1 if the literals' storage is shared.

- 15 EXAMPLE 7 Since compound literals are unnamed, a single compound literal cannot specify a circularly linked object. For example, there is no way to write a self-referential compound literal that could be used as the function argument in place of the named object `endless_zeros` below:

```
struct int_list { int car; struct int_list *cdr; };
struct int_list endless_zeros = {0, &endless_zeros};
eval(endless_zeros);
```

- 16 EXAMPLE 8 Each compound literal creates only a single object in a given scope:

```
struct s { int i; };

int f (void)
{
    struct s *p = 0, *q;
    int j = 0;

    again:
    q = p, p = &((struct s){ j++ });
    if (j < 2) goto again;

    return p == q && q->i == 1;
}
```

The function `f()` always returns the value 1.

- 17 Note that if an iteration statement were used instead of an explicit `goto` and a labeled statement, the lifetime of the unnamed object would be the body of the loop only, and on entry next time around `p` would have an indeterminate value, which would result in undefined behavior.

**Forward references:** type names (6.7.6), initialization (6.7.8).

### 6.5.3 Unary operators

#### Syntax

- 1 *unary-expression*:
- postfix-expression*
  - ++** *unary-expression*
  - *unary-expression*
  - unary-operator cast-expression*
  - sizeof** *unary-expression*
  - sizeof** ( *type-name* )
- unary-operator*: one of
- & \* + - ~ !**

#### 6.5.3.1 Prefix increment and decrement operators

##### Constraints

- 1 The operand of the prefix increment or decrement operator shall have qualified or unqualified real or pointer type and shall be a modifiable lvalue.

##### Semantics

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

**Forward references:** additive operators (6.5.6), compound assignment (6.5.16.2).

#### 6.5.3.2 Address and indirection operators

##### Constraints

- 1 The operand of the unary **&** operator shall be either a function designator, the result of a **[ ]** or unary **\*** operator, or an lvalue that designates an object that is not a bit-field and is not declared with the **register** storage-class specifier.
- 2 The operand of the unary **\*** operator shall have pointer type.

##### Semantics

- 3 The unary **&** operator yields the address of its operand. If the operand has type “*type*”, the result has type “pointer to *type*”. If the operand is the result of a unary **\*** operator, neither that operator nor the **&** operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. Similarly, if the operand is the result of a **[ ]** operator, neither the **&** operator nor

the unary `*` that is implied by the `[]` is evaluated and the result is as if the `&` operator were removed and the `[]` operator were changed to a `+` operator. Otherwise, the result is a pointer to the object or function designated by its operand.

- 4 The unary `*` operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type “pointer to *type*”, the result has type “*type*”. If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined.<sup>84)</sup>

**Forward references:** storage-class specifiers (6.7.1), structure and union specifiers (6.7.2.1).

### 6.5.3.3 Unary arithmetic operators

#### Constraints

- 1 The operand of the unary `+` or `-` operator shall have arithmetic type; of the `~` operator, integer type; of the `!` operator, scalar type.

#### Semantics

- 2 The result of the unary `+` operator is the value of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.
- 3 The result of the unary `-` operator is the negative of its (promoted) operand. The integer promotions are performed on the operand, and the result has the promoted type.
- 4 The result of the `~` operator is the bitwise complement of its (promoted) operand (that is, each bit in the result is set if and only if the corresponding bit in the converted operand is not set). The integer promotions are performed on the operand, and the result has the promoted type. If the promoted type is an unsigned type, the expression `~E` is equivalent to the maximum value representable in that type minus `E`.
- 5 The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. The result has type `int`. The expression `!E` is equivalent to `(0==E)`.

---

84) Thus, `&*E` is equivalent to `E` (even if `E` is a null pointer), and `&(E1[E2])` to `((E1)+(E2))`. It is always true that if `E` is a function designator or an lvalue that is a valid operand of the unary `&` operator, `*&E` is a function designator or an lvalue equal to `E`. If `*P` is an lvalue and `T` is the name of an object pointer type, `*(T)P` is an lvalue that has a type compatible with that to which `T` points.

Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

### 6.5.3.4 The **sizeof** operator

#### Constraints

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

#### Semantics

- 2 The **sizeof** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. If the type of the operand is a variable length array type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.
- 3 When applied to an operand that has type **char**, **unsigned char**, or **signed char**, (or a qualified version thereof) the result is 1. When applied to an operand that has array type, the result is the total number of bytes in the array.<sup>85)</sup> When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding.
- 4 The value of the result is implementation-defined, and its type (an unsigned integer type) is **size\_t**, defined in **<stddef.h>** (and other headers).
- 5 EXAMPLE 1 A principal use of the **sizeof** operator is in communication with routines such as storage allocators and I/O systems. A storage-allocation function might accept a size (in bytes) of an object to allocate and return a pointer to **void**. For example:

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

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

- 6 EXAMPLE 2 Another use of the **sizeof** operator is to compute the number of elements in an array:

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

- 7 EXAMPLE 3 In this example, the size of a variable length array is computed and returned from a function:

```
#include <stddef.h>
size_t fsize3(int n)
{
    char b[n+3]; // variable length array
    return sizeof b; // execution time sizeof
}
```

---

85) When applied to a parameter declared to have array or function type, the **sizeof** operator yields the size of the adjusted (pointer) type (see 6.9.1).

```

int main()
{
    size_t size;
    size = fsize3(10); // fsize3 returns 13
    return 0;
}

```

**Forward references:** common definitions `<stddef.h>` (7.17), declarations (6.7), structure and union specifiers (6.7.2.1), type names (6.7.6), array declarators (6.7.5.2).

## 6.5.4 Cast operators

### Syntax

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

### Constraints

- 2 Unless the type name specifies a void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type.
- 3 Conversions that involve pointers, other than where permitted by the constraints of 6.5.16.1, shall be specified by means of an explicit cast.

### Semantics

- 4 Preceding an expression by a parenthesized type name converts the value of the expression to the named type. This construction is called a *cast*.<sup>86)</sup> A cast that specifies no conversion has no effect on the type or value of an expression.<sup>87)</sup>

**Forward references:** equality operators (6.5.9), function declarators (including prototypes) (6.7.5.3), simple assignment (6.5.16.1), type names (6.7.6).

---

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

87) If the value of the expression is represented with greater precision or range than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type.

## 6.5.5 Multiplicative operators

### Syntax

- 1 *multiplicative-expression:*  
     *cast-expression*  
     *multiplicative-expression* \* *cast-expression*  
     *multiplicative-expression* / *cast-expression*  
     *multiplicative-expression* % *cast-expression*

### Constraints

- 2 Each of the operands shall have arithmetic type. The operands of the % operator shall have integer type.

### Semantics

- 3 The usual arithmetic conversions are performed on the operands.  
 4 The result of the binary \* operator is the product of the operands.  
 5 The result of the / operator is the quotient from the division of the first operand by the second; the result of the % operator is the remainder. In both operations, if the value of the second operand is zero, the behavior is undefined.  
 6 When integers are divided, the result of the / operator is the algebraic quotient with any fractional part discarded.<sup>88)</sup> If the quotient  $\mathbf{a/b}$  is representable, the expression  $(\mathbf{a/b}) * \mathbf{b} + \mathbf{a \% b}$  shall equal  $\mathbf{a}$ .

## 6.5.6 Additive operators

### Syntax

- 1 *additive-expression:*  
     *multiplicative-expression*  
     *additive-expression* + *multiplicative-expression*  
     *additive-expression* - *multiplicative-expression*

### Constraints

- 2 For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to an object type and the other shall have integer type. (Incrementing is equivalent to adding 1.)  
 3 For subtraction, one of the following shall hold:  
 — both operands have arithmetic type;

---

88) This is often called “truncation toward zero”.

- both operands are pointers to qualified or unqualified versions of compatible object types; or
- the left operand is a pointer to an object type and the right operand has integer type.

(Decrementing is equivalent to subtracting 1.)

### Semantics

- 4 If both operands have arithmetic type, the usual arithmetic conversions are performed on them.
- 5 The result of the binary `+` operator is the sum of the operands.
- 6 The result of the binary `-` operator is the difference resulting from the subtraction of the second operand from the first.
- 7 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.
- 8 When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression `P` points to the  $i$ -th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value  $n$ ) point to, respectively, the  $i+n$ -th and  $i-n$ -th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary `*` operator that is evaluated.
- 9 When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is `ptrdiff_t` defined in the `<stddef.h>` header. If the result is not representable in an object of that type, the behavior is undefined. In other words, if the expressions `P` and `Q` point to, respectively, the  $i$ -th and  $j$ -th elements of an array object, the expression `(P)-(Q)` has the value  $i-j$  provided the value fits in an object of type `ptrdiff_t`. Moreover, if the expression `P` points either to an element of an array object or one past the last element of an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same

value as  $((Q)-(P))+1$  and as  $-((P)-((Q)+1))$ , and has the value zero if the expression  $P$  points one past the last element of the array object, even though the expression  $(Q)+1$  does not point to an element of the array object.<sup>89)</sup>

- 10 EXAMPLE Pointer arithmetic is well defined with pointers to variable length array types.

```

{
    int n = 4, m = 3;
    int a[n][m];
    int (*p)[m] = a; // p == &a[0]
    p += 1;         // p == &a[1]
    (*p)[2] = 99;   // a[1][2] == 99
    n = p - a;      // n == 1
}

```

- 11 If array  $a$  in the above example were declared to be an array of known constant size, and pointer  $p$  were declared to be a pointer to an array of the same known constant size (pointing to  $a$ ), the results would be the same.

**Forward references:** array declarators (6.7.5.2), common definitions `<stddef.h>` (7.17).

## 6.5.7 Bitwise shift operators

### Syntax

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

### Constraints

- 2 Each of the operands shall have integer type.

### Semantics

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

---

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

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

- 4 The result of  $\mathbf{E1} \ll \mathbf{E2}$  is  $\mathbf{E1}$  left-shifted  $\mathbf{E2}$  bit positions; vacated bits are filled with zeros. If  $\mathbf{E1}$  has an unsigned type, the value of the result is  $\mathbf{E1} \times 2^{\mathbf{E2}}$ , reduced modulo one more than the maximum value representable in the result type. If  $\mathbf{E1}$  has a signed type and nonnegative value, and  $\mathbf{E1} \times 2^{\mathbf{E2}}$  is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined.
- 5 The result of  $\mathbf{E1} \gg \mathbf{E2}$  is  $\mathbf{E1}$  right-shifted  $\mathbf{E2}$  bit positions. If  $\mathbf{E1}$  has an unsigned type or if  $\mathbf{E1}$  has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of  $\mathbf{E1} / 2^{\mathbf{E2}}$ . If  $\mathbf{E1}$  has a signed type and a negative value, the resulting value is implementation-defined.

### 6.5.8 Relational operators

#### Syntax

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

#### Constraints

- 2 One of the following shall hold:
- both operands have real type;
  - both operands are pointers to qualified or unqualified versions of compatible object types; or
  - both operands are pointers to qualified or unqualified versions of compatible incomplete types.

#### Semantics

- 3 If both of the operands have arithmetic type, the usual arithmetic conversions are performed.
- 4 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.
- 5 When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to. If two pointers to object or incomplete types both point to the same object, or both point one past the last element of the same array object, they compare equal. If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript

values compare greater than pointers to elements of the same array with lower subscript values. All pointers to members of the same union object compare equal. If the expression **P** points to an element of an array object and the expression **Q** points to the last element of the same array object, the pointer expression **Q+1** compares greater than **P**. In all other cases, the behavior is undefined.

- 6 Each of the operators **<** (less than), **>** (greater than), **<=** (less than or equal to), and **>=** (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.<sup>90)</sup> The result has type **int**.

### 6.5.9 Equality operators

#### Syntax

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

#### Constraints

- 2 One of the following shall hold:
- both operands have arithmetic type;
  - both operands are pointers to qualified or unqualified versions of compatible types;
  - one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**; or
  - one operand is a pointer and the other is a null pointer constant.

#### Semantics

- 3 The **==** (equal to) and **!=** (not equal to) operators are analogous to the relational operators except for their lower precedence.<sup>91)</sup> Each of the operators yields 1 if the specified relation is true and 0 if it is false. The result has type **int**. For any pair of operands, exactly one of the relations is true.
- 4 If both of the operands have arithmetic type, the usual arithmetic conversions are performed. Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal.

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

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

- 5 Otherwise, at least one operand is a pointer. If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. If one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**, the former is converted to the type of the latter.
- 6 Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.<sup>92)</sup>
- 7 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

### 6.5.10 Bitwise AND operator

#### Syntax

- 1 *AND-expression:*  
*equality-expression*  
*AND-expression & equality-expression*

#### Constraints

- 2 Each of the operands shall have integer type.

#### Semantics

- 3 The usual arithmetic conversions are performed on the operands.
- 4 The result of the binary **&** operator is the bitwise AND of the operands (that is, each bit in the result is set if and only if each of the corresponding bits in the converted operands is set).

---

92) Two objects may be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior.

### 6.5.11 Bitwise exclusive OR operator

#### Syntax

- 1 *exclusive-OR-expression:*  
*AND-expression*  
*exclusive-OR-expression*  $\wedge$  *AND-expression*

#### Constraints

- 2 Each of the operands shall have integer type.

#### Semantics

- 3 The usual arithmetic conversions are performed on the operands.  
4 The result of the  $\wedge$  operator is the bitwise exclusive OR of the operands (that is, each bit in the result is set if and only if exactly one of the corresponding bits in the converted operands is set).

### 6.5.12 Bitwise inclusive OR operator

#### Syntax

- 1 *inclusive-OR-expression:*  
*exclusive-OR-expression*  
*inclusive-OR-expression*  $\mid$  *exclusive-OR-expression*

#### Constraints

- 2 Each of the operands shall have integer type.

#### Semantics

- 3 The usual arithmetic conversions are performed on the operands.  
4 The result of the  $\mid$  operator is the bitwise inclusive OR of the operands (that is, each bit in the result is set if and only if at least one of the corresponding bits in the converted operands is set).

### 6.5.13 Logical AND operator

#### Syntax

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

#### Constraints

- 2 Each of the operands shall have scalar type.

#### Semantics

- 3 The **&&** operator shall yield 1 if both of its operands compare unequal to 0; otherwise, it yields 0. The result has type **int**.
- 4 Unlike the bitwise binary **&** operator, the **&&** operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares equal to 0, the second operand is not evaluated.

### 6.5.14 Logical OR operator

#### Syntax

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

#### Constraints

- 2 Each of the operands shall have scalar type.

#### Semantics

- 3 The **||** operator shall yield 1 if either of its operands compare unequal to 0; otherwise, it yields 0. The result has type **int**.
- 4 Unlike the bitwise **|** operator, the **||** operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares unequal to 0, the second operand is not evaluated.

## 6.5.15 Conditional operator

### Syntax

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

### Constraints

- 2    The first operand shall have scalar type.
- 3    One of the following shall hold for the second and third operands:
- both operands have arithmetic type;
  - both operands have the same structure or union type;
  - both operands have void type;
  - both operands are pointers to qualified or unqualified versions of compatible types;
  - one operand is a pointer and the other is a null pointer constant; or
  - one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**.

### Semantics

- 4    The first operand is evaluated; there is a sequence point after its evaluation. The second operand is evaluated only if the first compares unequal to 0; the third operand is evaluated only if the first compares equal to 0; the result is the value of the second or third operand (whichever is evaluated), converted to the type described below.<sup>93)</sup> If an attempt is made to modify the result of a conditional operator or to access it after the next sequence point, the behavior is undefined.
- 5    If both the second and third operands have arithmetic type, the result type that would be determined by the usual arithmetic conversions, were they applied to those two operands, is the type of the result. If both the operands have structure or union type, the result has that type. If both operands have void type, the result has void type.
- 6    If both the second and third operands are pointers or one is a null pointer constant and the other is a pointer, the result type is a pointer to a type qualified with all the type qualifiers of the types pointed-to by both operands. Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type; if one operand is a null pointer constant, the result has the type of the other operand; otherwise, one operand is a pointer to **void** or a qualified version of **void**, in which case the result type is a

---

93) A conditional expression does not yield an lvalue.

pointer to an appropriately qualified version of **void**.

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

- 8 Given the declarations

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

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

|      |      |                      |
|------|------|----------------------|
| c_vp | c_ip | const void *         |
| v_ip | 0    | volatile int *       |
| c_ip | v_ip | const volatile int * |
| vp   | c_cp | const void *         |
| ip   | c_ip | const int *          |
| vp   | ip   | void *               |

## 6.5.16 Assignment operators

### Syntax

- 1 *assignment-expression*:  
*conditional-expression*  
*unary-expression assignment-operator assignment-expression*
- assignment-operator*: one of
- =   \*=   /=   %=   +=   -=   <<=   >>=   &=   ^=   |=**

### Constraints

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

### Semantics

- 3 An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. The type of an assignment expression is the type of the left operand unless the left operand has qualified type, in which case it is the unqualified version of the type of the left operand. The side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point.
- 4 The order of evaluation of the operands is unspecified. If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined.

### 6.5.16.1 Simple assignment

#### Constraints

- 1 One of the following shall hold:<sup>94)</sup>
  - the left operand has qualified or unqualified arithmetic type and the right has arithmetic type;
  - the left operand has a qualified or unqualified version of a structure or union type compatible with the type of the right;
  - both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;
  - one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;
  - the left operand is a pointer and the right is a null pointer constant; or
  - the left operand has type **\_Bool** and the right is a pointer.

#### Semantics

- 2 In *simple assignment* (**=**), the value of the right operand is converted to the type of the assignment expression and replaces the value stored in the object designated by the left operand.
- 3 If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; otherwise, the behavior is undefined.
- 4 EXAMPLE 1 In the program fragment

```

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

```

the **int** value returned by the function may be truncated when stored in the **char**, and then converted back to **int** width prior to the comparison. In an implementation in which “plain” **char** has the same range of values as **unsigned char** (and **char** is narrower than **int**), the result of the conversion cannot be

---

94) The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to “the value of the expression” and thus removes any type qualifiers that were applied to the type category of the expression (for example, it removes **const** but not **volatile** from the type **int volatile \* const**).

negative, so the operands of the comparison can never compare equal. Therefore, for full portability, the variable `c` should be declared as `int`.

- 5 EXAMPLE 2 In the fragment:

```
char c;
int i;
long l;

l = (c = i);
```

the value of `i` is converted to the type of the assignment expression `c = i`, that is, `char` type. The value of the expression enclosed in parentheses is then converted to the type of the outer assignment expression, that is, `long int` type.

- 6 EXAMPLE 3 Consider the fragment:

```
const char **cpp;
char *p;
const char c = 'A';

cpp = &p;           // constraint violation
*cpp = &c;         // valid
*p = 0;           // valid
```

The first assignment is unsafe because it would allow the following valid code to attempt to change the value of the const object `c`.

## 6.5.16.2 Compound assignment

### Constraints

- 1 For the operators `+=` and `-=` only, either the left operand shall be a pointer to an object type and the right shall have integer type, or the left operand shall have qualified or unqualified arithmetic type and the right shall have arithmetic type.
- 2 For the other operators, each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator.

### Semantics

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

## 6.5.17 Comma operator

### Syntax

- 1 *expression:*  
*assignment-expression*  
*expression* , *assignment-expression*

### Semantics

- 2 The left operand of a comma operator is evaluated as a void expression; there is a sequence point after its evaluation. Then the right operand is evaluated; the result has its type and value.<sup>95)</sup> If an attempt is made to modify the result of a comma operator or to access it after the next sequence point, the behavior is undefined.
- 3 **EXAMPLE** As indicated by the syntax, the comma operator (as described in this subclause) cannot appear in contexts where a comma is used to separate items in a list (such as arguments to functions or lists of initializers). On the other hand, it can be used within a parenthesized expression or within the second expression of a conditional operator in such contexts. In the function call

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

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

**Forward references:** initialization (6.7.8).

---

95) A comma operator does not yield an lvalue.

## 6.6 Constant expressions

### Syntax

- 1            *constant-expression:*  
              *conditional-expression*

### Description

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

### Constraints

- 3     Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.<sup>96)</sup>
- 4     Each constant expression shall evaluate to a constant that is in the range of representable values for its type.

### Semantics

- 5     An expression that evaluates to a constant is required in several contexts. If a floating expression is evaluated in the translation environment, the arithmetic precision and range shall be at least as great as if the expression were being evaluated in the execution environment.
- 6     An *integer constant expression*<sup>97)</sup> shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, and floating constants that are the immediate operands of casts. Cast operators in an integer constant expression shall only convert arithmetic types to integer types, except as part of an operand to the **sizeof** operator.
- 7     More latitude is permitted for constant expressions in initializers. Such a constant expression shall be, or evaluate to, one of the following:
- an arithmetic constant expression,
  - a null pointer constant,

---

96) The operand of a **sizeof** operator is usually not evaluated (6.5.3.4).

97) An integer constant expression is used to specify the size of a bit-field member of a structure, the value of an enumeration constant, the size of an array, or the value of a **case** constant. Further constraints that apply to the integer constant expressions used in conditional-inclusion preprocessing directives are discussed in 6.10.1.

- an address constant, or
  - an address constant for an object type plus or minus an integer constant expression.
- 8 An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, and **sizeof** expressions. Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to a **sizeof** operator whose result is an integer constant.
- 9 An *address constant* is a null pointer, a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; it shall be created explicitly using the unary **&** operator or an integer constant cast to pointer type, or implicitly by the use of an expression of array or function type. The array-subscript **[ ]** and member-access **.** and **->** operators, the address **&** and indirection **\*** unary operators, and pointer casts may be used in the creation of an address constant, but the value of an object shall not be accessed by use of these operators.
- 10 An implementation may accept other forms of constant expressions.
- 11 The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.<sup>98)</sup>

**Forward references:** array declarators (6.7.5.2), initialization (6.7.8).

---

98) Thus, in the following initialization,

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

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

## 6.7 Declarations

### Syntax

- 1 *declaration:*  
       *declaration-specifiers init-declarator-list*<sub>opt</sub> **;**
- declaration-specifiers:*  
       *storage-class-specifier declaration-specifiers*<sub>opt</sub>  
       *type-specifier declaration-specifiers*<sub>opt</sub>  
       *type-qualifier declaration-specifiers*<sub>opt</sub>  
       *function-specifier declaration-specifiers*<sub>opt</sub>
- init-declarator-list:*  
       *init-declarator*  
       *init-declarator-list* **,** *init-declarator*
- init-declarator:*  
       *declarator*  
       *declarator* **=** *initializer*

### Constraints

- 2 A declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration.
- 3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified in 6.7.2.3.
- 4 All declarations in the same scope that refer to the same object or function shall specify compatible types.

### Semantics

- 5 A declaration specifies the interpretation and attributes of a set of identifiers. A *definition* of an identifier is a declaration for that identifier that:
- for an object, causes storage to be reserved for that object;
  - for a function, includes the function body;<sup>99)</sup>
  - for an enumeration constant or typedef name, is the (only) declaration of the identifier.
- 6 The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. The *init-declarator-list* is a comma-separated sequence of declarators, each of which may have

---

99) Function definitions have a different syntax, described in 6.9.1.

additional type information, or an initializer, or both. The declarators contain the identifiers (if any) being declared.

- 7 If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer; in the case of function parameters (including in prototypes), it is the adjusted type (see 6.7.5.3) that is required to be complete.

**Forward references:** declarators (6.7.5), enumeration specifiers (6.7.2.2), initialization (6.7.8).

## 6.7.1 Storage-class specifiers

### Syntax

- 1 *storage-class-specifier:*  
     **typedef**  
     **extern**  
     **static**  
     **auto**  
     **register**

### Constraints

- 2 At most, one storage-class specifier may be given in the declaration specifiers in a declaration.<sup>100)</sup>

### Semantics

- 3 The **typedef** specifier is called a “storage-class specifier” for syntactic convenience only; it is discussed in 6.7.7. The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4.
- 4 A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. The extent to which such suggestions are effective is implementation-defined.<sup>101)</sup>
- 5 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **extern**.

---

100) See “future language directions” (6.11.5).

101) The implementation may treat any **register** declaration simply as an **auto** declaration. However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier **register** cannot be computed, either explicitly (by use of the unary **&** operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1). Thus, the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof**.

- 6 If an aggregate or union object is declared with a storage-class specifier other than **typedef**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

**Forward references:** type definitions (6.7.7).

## 6.7.2 Type specifiers

### Syntax

- 1 *type-specifier:*
- void**
  - char**
  - short**
  - int**
  - long**
  - float**
  - double**
  - signed**
  - unsigned**
  - \_Bool**
  - \_Complex**
  - struct-or-union-specifier* \*
  - enum-specifier*
  - typedef-name*

### Constraints

- 2 At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct declaration and type name. Each list of type specifiers shall be one of the following sets (delimited by commas, when there is more than one set on a line); the type specifiers may occur in any order, possibly intermixed with the other declaration specifiers.

- **void**
- **char**
- **signed char**
- **unsigned char**
- **short, signed short, short int, or signed short int**
- **unsigned short, or unsigned short int**
- **int, signed, or signed int**

- **unsigned**, or **unsigned int**
- **long**, **signed long**, **long int**, or **signed long int**
- **unsigned long**, or **unsigned long int**
- **long long**, **signed long long**, **long long int**, or **signed long long int**
- **unsigned long long**, or **unsigned long long int**
- **float**
- **double**
- **long double**
- **\_Bool**
- **float \_Complex**
- **double \_Complex**
- **long double \_Complex**
- struct or union specifier \*
- enum specifier
- typedef name

- 3 The type specifier **\_Complex** shall not be used if the implementation does not provide complex types.<sup>102)</sup> |

### Semantics

- 4 Specifiers for structures, unions, and enumerations are discussed in 6.7.2.1 through 6.7.2.3. Declarations of typedef names are discussed in 6.7.7. The characteristics of the other types are discussed in 6.2.5.
- 5 Each of the comma-separated sets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier **int** designates the same type as **signed int** or the same type as **unsigned int**.

**Forward references:** enumeration specifiers (6.7.2.2), structure and union specifiers (6.7.2.1), tags (6.7.2.3), type definitions (6.7.7).

---

102) Freestanding implementations are not required to provide complex types. \*

### 6.7.2.1 Structure and union specifiers

#### Syntax

- 1 *struct-or-union-specifier*:
- ```

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

```
- struct-or-union*:
- ```

    struct
    union

```
- struct-declaration-list*:
- ```

    struct-declaration
    struct-declaration-list struct-declaration

```
- struct-declaration*:
- ```

    specifier-qualifier-list struct-declarator-list ;

```
- specifier-qualifier-list*:
- ```

    type-specifier specifier-qualifier-listopt
    type-qualifier specifier-qualifier-listopt

```
- struct-declarator-list*:
- ```

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

```
- struct-declarator*:
- ```

    declarator
    declaratoropt : constant-expression

```

#### Constraints

- 2 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.
- 3 The expression that specifies the width of a bit-field shall be an integer constant expression with a nonnegative value that does not exceed the width of an object of the type that would be specified were the colon and expression omitted. If the value is zero, the declaration shall have no declarator.
- 4 A bit-field shall have a type that is a qualified or unqualified version of **\_Bool**, **signed int**, **unsigned int**, or some other implementation-defined type.

## Semantics

- 5 As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.
- 6 Structure and union specifiers have the same form.
- 7 The presence of a struct-declaration-list in a struct-or-union-specifier declares a new type, within a translation unit. The struct-declaration-list is a sequence of declarations for the members of the structure or union. If the struct-declaration-list contains no named members, the behavior is undefined. The type is incomplete until after the `}` that terminates the list.
- 8 A member of a structure or union may have any object type other than a variably modified type.<sup>103)</sup> In addition, a member may be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a *bit-field*;<sup>104)</sup> its width is preceded by a colon.
- 9 A bit-field is interpreted as a signed or unsigned integer type consisting of the specified number of bits.<sup>105)</sup> If the value 0 or 1 is stored into a nonzero-width bit-field of type `_Bool`, the value of the bit-field shall compare equal to the value stored.
- 10 An implementation may allocate any addressable storage unit large enough to hold a bit-field. If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined. The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined. The alignment of the addressable storage unit is unspecified.
- 11 A bit-field declaration with no declarator, but only a colon and a width, indicates an unnamed bit-field.<sup>106)</sup> As a special case, a bit-field structure member with a width of 0 indicates that no further bit-field is to be packed into the unit in which the previous bit-field, if any, was placed.

---

103) A structure or union can not contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3.

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

105) As specified in 6.7.2 above, if the actual type specifier used is `int` or a typedef-name defined as `int`, then it is implementation-defined whether the bit-field is signed or unsigned.

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

- 12 Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type.
- 13 Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa. There may be unnamed padding within a structure object, but not at its beginning.
- 14 The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.
- 15 There may be unnamed padding at the end of a structure or union.
- 16 As a special case, the last element of a structure with more than one named member may have an incomplete array type; this is called a *flexible array member*. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply. However, when a `.` (or `->`) operator has a left operand that is (a pointer to) a structure with a flexible array member and the right operand names that member, it behaves as if that member were replaced with the longest array (with the same element type) that would not make the structure larger than the object being accessed; the offset of the array shall remain that of the flexible array member, even if this would differ from that of the replacement array. If this array would have no elements, it behaves as if it had one element but the behavior is undefined if any attempt is made to access that element or to generate a pointer one past it.
- 17 EXAMPLE After the declaration:
- ```
    struct s { int n; double d[]; };
```
- the structure `struct s` has a flexible array member `d`. A typical way to use this is:
- ```
    int m = /* some value */;
    struct s *p = malloc(sizeof (struct s) + sizeof (double [m]));
```
- and assuming that the call to `malloc` succeeds, the object pointed to by `p` behaves, for most purposes, as if `p` had been declared as:
- ```
    struct { int n; double d[m]; } *p;
```
- (there are circumstances in which this equivalence is broken; in particular, the offsets of member `d` might not be the same).
- 18 Following the above declaration:

```

struct s t1 = { 0 };           // valid
struct s t2 = { 1, { 4.2 } }; // invalid
t1.n = 4;                     // valid
t1.d[0] = 4.2;                // might be undefined behavior

```

The initialization of **t2** is invalid (and violates a constraint) because **struct s** is treated as if it did not contain member **d**. The assignment to **t1.d[0]** is probably undefined behavior, but it is possible that

```

sizeof (struct s) >= offsetof(struct s, d) + sizeof (double)

```

in which case the assignment would be legitimate. Nevertheless, it cannot appear in strictly conforming code.

- 19 After the further declaration:

```

struct ss { int n; };

```

the expressions:

```

sizeof (struct s) >= sizeof (struct ss)
sizeof (struct s) >= offsetof(struct s, d)

```

are always equal to 1.

- 20 If **sizeof (double)** is 8, then after the following code is executed:

```

struct s *s1;
struct s *s2;
s1 = malloc(sizeof (struct s) + 64);
s2 = malloc(sizeof (struct s) + 46);

```

and assuming that the calls to **malloc** succeed, the objects pointed to by **s1** and **s2** behave, for most purposes, as if the identifiers had been declared as:

```

struct { int n; double d[8]; } *s1;
struct { int n; double d[5]; } *s2;

```

- 21 Following the further successful assignments:

```

s1 = malloc(sizeof (struct s) + 10);
s2 = malloc(sizeof (struct s) + 6);

```

they then behave as if the declarations were:

```

struct { int n; double d[1]; } *s1, *s2;

```

and:

```

double *dp;
dp = &(s1->d[0]); // valid
*dp = 42; // valid
dp = &(s2->d[0]); // valid
*dp = 42; // undefined behavior

```

- 22 The assignment:

```

*s1 = *s2;

```

only copies the member **n**; if any of the array elements are within the first **sizeof (struct s)** bytes of the structure, they might be copied or simply overwritten with indeterminate values.

**Forward references:** tags (6.7.2.3).

### 6.7.2.2 Enumeration specifiers

#### Syntax

- 1 *enum-specifier*:
- ```

enum identifieropt { enumerator-list }
enum identifieropt { enumerator-list , }
enum identifier

```
- enumerator-list*:
- ```

enumerator
enumerator-list , enumerator

```
- enumerator*:
- ```

enumeration-constant
enumeration-constant = constant-expression

```

#### Constraints

- 2 The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an **int**.

#### Semantics

- 3 The identifiers in an enumerator list are declared as constants that have type **int** and may appear wherever such are permitted.<sup>107)</sup> An enumerator with = defines its enumeration constant as the value of the constant expression. If the first enumerator has no =, the value of its enumeration constant is 0. Each subsequent enumerator with no = defines its enumeration constant as the value of the constant expression obtained by adding 1 to the value of the previous enumeration constant. (The use of enumerators with = may produce enumeration constants with values that duplicate other values in the same enumeration.) The enumerators of an enumeration are also known as its members.
- 4 Each enumerated type shall be compatible with **char**, a signed integer type, or an unsigned integer type. The choice of type is implementation-defined,<sup>108)</sup> but shall be capable of representing the values of all the members of the enumeration. The enumerated type is incomplete until after the } that terminates the list of enumerator declarations.

---

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

108) An implementation may delay the choice of which integer type until all enumeration constants have been seen.

- 5 EXAMPLE The following fragment:

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

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

**Forward references:** tags (6.7.2.3).

### 6.7.2.3 Tags

#### Constraints

- 1 A specific type shall have its content defined at most once.
- 2 A type specifier of the form

```
enum identifier
```

without an enumerator list shall only appear after the type it specifies is complete.

#### Semantics

- 3 All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type. The type is incomplete<sup>109)</sup> until the closing brace of the list defining the content, and complete thereafter.
- 4 Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type.
- 5 A type specifier of the form

```
struct-or-union identifieropt { struct-declaration-list }
```

or

```
enum identifier { enumerator-list }
```

or

```
enum identifier { enumerator-list , }
```

declares a structure, union, or enumerated type. The list defines the *structure content*, *union content*, or *enumeration content*. If an identifier is provided,<sup>110)</sup> the type specifier also declares the identifier to be the tag of that type.

---

109) An incomplete type may only be used when the size of an object of that type is not needed. It is not needed, for example, when a typedef name is declared to be a specifier for a structure or union, or when a pointer to or a function returning a structure or union is being declared. (See incomplete types in 6.2.5.) The specification has to be complete before such a function is called or defined.

- 6 A declaration of the form

*struct-or-union identifier ;*

specifies a structure or union type and declares the identifier as a tag of that type.<sup>110)</sup>

- 7 If a type specifier of the form

*struct-or-union identifier*

occurs other than as part of one of the above forms, and no other declaration of the identifier as a tag is visible, then it declares an incomplete structure or union type, and declares the identifier as the tag of that type.<sup>111)</sup>

- 8 If a type specifier of the form

*struct-or-union identifier*

or

**enum** *identifier*

occurs other than as part of one of the above forms, and a declaration of the identifier as a tag is visible, then it specifies the same type as that other declaration, and does not redeclare the tag.

- 9 EXAMPLE 1 This mechanism allows declaration of a self-referential structure.

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

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

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

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

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

---

110) If there is no identifier, the type can, within the translation unit, only be referred to by the declaration of which it is a part. Of course, when the declaration is of a typedef name, subsequent declarations can make use of that typedef name to declare objects having the specified structure, union, or enumerated type.

111) A similar construction with **enum** does not exist.

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

- 11 EXAMPLE 2 To illustrate the use of prior declaration of a tag to specify a pair of mutually referential structures, the declarations

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

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

```
struct s2;
```

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

**Forward references:** declarators (6.7.5), array declarators (6.7.5.2), type definitions (6.7.7).

### 6.7.3 Type qualifiers

#### Syntax

- 1 *type-qualifier:*
- ```
    const
    restrict
    volatile
```

#### Constraints

- 2 Types other than pointer types derived from object or incomplete types shall not be restrict-qualified.

#### Semantics

- 3 The properties associated with qualified types are meaningful only for expressions that are lvalues.<sup>112)</sup>
- 4 If the same qualifier appears more than once in the same *specifier-qualifier-list*, either directly or via one or more **typedefs**, the behavior is the same as if it appeared only once.

---

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

- 5 If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.<sup>113)</sup>
- 6 An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously.<sup>114)</sup> What constitutes an access to an object that has volatile-qualified type is implementation-defined.
- 7 An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association, defined in 6.7.3.1 below, requires that all accesses to that object use, directly or indirectly, the value of that particular pointer.<sup>115)</sup> The intended use of the **restrict** qualifier (like the **register** storage class) is to promote optimization, and deleting all instances of the qualifier from all preprocessing translation units composing a conforming program does not change its meaning (i.e., observable behavior).
- 8 If the specification of an array type includes any type qualifiers, the element type is so-qualified, not the array type. If the specification of a function type includes any type qualifiers, the behavior is undefined.<sup>116)</sup>
- 9 For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type.
- 10 EXAMPLE 1 An object declared

```
extern const volatile int real_time_clock;
```

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

---

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

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

115) For example, a statement that assigns a value returned by **malloc** to a single pointer establishes this association between the allocated object and the pointer.

116) Both of these can occur through the use of **typedefs**.

- 11 EXAMPLE 2 The following declarations and expressions illustrate the behavior when type qualifiers modify an aggregate type:

```

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

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

```

### 6.7.3.1 Formal definition of restrict

- 1 Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a restrict-qualified pointer to type **T**.
- 2 If **D** appears inside a block and does not have storage class **extern**, let **B** denote the block. If **D** appears in the list of parameter declarations of a function definition, let **B** denote the associated block. Otherwise, let **B** denote the block of **main** (or the block of whatever function is called at program startup in a freestanding environment).
- 3 In what follows, a pointer expression **E** is said to be *based* on object **P** if (at some sequence point in the execution of **B** prior to the evaluation of **E**) modifying **P** to point to a copy of the array object into which it formerly pointed would change the value of **E**.<sup>117)</sup> Note that “based” is defined only for expressions with pointer types.
- 4 During each execution of **B**, let **L** be any lvalue that has **&L** based on **P**. If **L** is used to access the value of the object **X** that it designates, and **X** is also modified (by any means), then the following requirements apply: **T** shall not be const-qualified. Every other lvalue used to access the value of **X** shall also have its address based on **P**. Every access that modifies **X** shall be considered also to modify **P**, for the purposes of this subclause. If **P** is assigned the value of a pointer expression **E** that is based on another restricted pointer object **P2**, associated with block **B2**, then either the execution of **B2** shall begin before the execution of **B**, or the execution of **B2** shall end prior to the assignment. If these requirements are not met, then the behavior is undefined.
- 5 Here an execution of **B** means that portion of the execution of the program that would correspond to the lifetime of an object with scalar type and automatic storage duration

---

117) In other words, **E** depends on the value of **P** itself rather than on the value of an object referenced indirectly through **P**. For example, if identifier **p** has type **(int \*\*restrict)**, then the pointer expressions **p** and **p+1** are based on the restricted pointer object designated by **p**, but the pointer expressions **\*p** and **p[1]** are not.

associated with **B**.

- 6 A translator is free to ignore any or all aliasing implications of uses of **restrict**.

- 7 EXAMPLE 1 The file scope declarations

```
int * restrict a;
int * restrict b;
extern int c[];
```

assert that if an object is accessed using one of **a**, **b**, or **c**, and that object is modified anywhere in the program, then it is never accessed using either of the other two.

- 8 EXAMPLE 2 The function parameter declarations in the following example

```
void f(int n, int * restrict p, int * restrict q)
{
    while (n-- > 0)
        *p++ = *q++;
}
```

assert that, during each execution of the function, if an object is accessed through one of the pointer parameters, then it is not also accessed through the other.

- 9 The benefit of the **restrict** qualifiers is that they enable a translator to make an effective dependence analysis of function **f** without examining any of the calls of **f** in the program. The cost is that the programmer has to examine all of those calls to ensure that none give undefined behavior. For example, the second call of **f** in **g** has undefined behavior because each of **d[1]** through **d[49]** is accessed through both **p** and **q**.

```
void g(void)
{
    extern int d[100];
    f(50, d + 50, d); // valid
    f(50, d + 1, d); // undefined behavior
}
```

- 10 EXAMPLE 3 The function parameter declarations

```
void h(int n, int * restrict p, int * restrict q, int * restrict r)
{
    int i;
    for (i = 0; i < n; i++)
        p[i] = q[i] + r[i];
}
```

illustrate how an unmodified object can be aliased through two restricted pointers. In particular, if **a** and **b** are disjoint arrays, a call of the form **h(100, a, b, b)** has defined behavior, because array **b** is not modified within function **h**.

- 11 EXAMPLE 4 The rule limiting assignments between restricted pointers does not distinguish between a function call and an equivalent nested block. With one exception, only “outer-to-inner” assignments between restricted pointers declared in nested blocks have defined behavior.

```

{
    int * restrict p1;
    int * restrict q1;
    p1 = q1; // undefined behavior
    {
        int * restrict p2 = p1; // valid
        int * restrict q2 = q1; // valid
        p1 = q2; // undefined behavior
        p2 = q2; // undefined behavior
    }
}

```

- 12 The one exception allows the value of a restricted pointer to be carried out of the block in which it (or, more precisely, the ordinary identifier used to designate it) is declared when that block finishes execution. For example, this permits `new_vector` to return a `vector`.

```

typedef struct { int n; float * restrict v; } vector;
vector new_vector(int n)
{
    vector t;
    t.n = n;
    t.v = malloc(n * sizeof (float));
    return t;
}

```

## 6.7.4 Function specifiers

### Syntax

- 1 *function-specifier*:  
**inline**

### Constraints

- 2 Function specifiers shall be used only in the declaration of an identifier for a function.
- 3 An inline definition of a function with external linkage shall not contain a definition of a modifiable object with static storage duration, and shall not contain a reference to an identifier with internal linkage.
- 4 In a hosted environment, the **inline** function specifier shall not appear in a declaration of **main**.

### Semantics

- 5 A function declared with an **inline** function specifier is an *inline function*. The function specifier may appear more than once; the behavior is the same as if it appeared only once. Making a function an inline function suggests that calls to the function be as fast as possible.<sup>118)</sup> The extent to which such suggestions are effective is implementation-defined.<sup>119)</sup>
- 6 Any function with internal linkage can be an inline function. For a function with external linkage, the following restrictions apply: If a function is declared with an **inline**

function specifier, then it shall also be defined in the same translation unit. If all of the file scope declarations for a function in a translation unit include the **inline** function specifier without **extern**, then the definition in that translation unit is an *inline definition*. An inline definition does not provide an external definition for the function, and does not forbid an external definition in another translation unit. An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.<sup>120)</sup>

- 7 EXAMPLE The declaration of an inline function with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with **extern** creates an external definition. The following example shows an entire translation unit.

```

inline double fahr(double t)
{
    return (9.0 * t) / 5.0 + 32.0;
}

inline double cels(double t)
{
    return (5.0 * (t - 32.0)) / 9.0;
}

extern double fahr(double);    // creates an external definition

double convert(int is_fahr, double temp)
{
    /* A translator may perform inline substitutions */
    return is_fahr ? cels(temp) : fahr(temp);
}

```

- 8 Note that the definition of **fahr** is an external definition because **fahr** is also declared with **extern**, but the definition of **cels** is an inline definition. Because **cels** has external linkage and is referenced, an external definition has to appear in another translation unit (see 6.9); the inline definition and the external definition are distinct and either may be used for the call.

**Forward references:** function definitions (6.9.1).

---

118) By using, for example, an alternative to the usual function call mechanism, such as “inline substitution”. Inline substitution is not textual substitution, nor does it create a new function. Therefore, for example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called; and identifiers refer to the declarations in scope where the body occurs. Likewise, the function has a single address, regardless of the number of inline definitions that occur in addition to the external definition.

119) For example, an implementation might never perform inline substitution, or might only perform inline substitutions to calls in the scope of an **inline** declaration.

120) Since an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units, all corresponding objects with static storage duration are also distinct in each of the definitions.

## 6.7.5 Declarators

### Syntax

- 1 *declarator*:
- pointer*<sub>opt</sub> *direct-declarator*
- direct-declarator*:
- identifier*
- ( *declarator* )
- direct-declarator* [ *type-qualifier-list*<sub>opt</sub> *assignment-expression*<sub>opt</sub> ]
- direct-declarator* [ **static** *type-qualifier-list*<sub>opt</sub> *assignment-expression* ]
- direct-declarator* [ *type-qualifier-list* **static** *assignment-expression* ]
- direct-declarator* [ *type-qualifier-list*<sub>opt</sub> \* ]
- direct-declarator* ( *parameter-type-list* )
- direct-declarator* ( *identifier-list*<sub>opt</sub> )
- pointer*:
- \* *type-qualifier-list*<sub>opt</sub>
- \* *type-qualifier-list*<sub>opt</sub> *pointer*
- type-qualifier-list*:
- type-qualifier*
- type-qualifier-list* *type-qualifier*
- parameter-type-list*:
- parameter-list*
- parameter-list* , ...
- parameter-list*:
- parameter-declaration*
- parameter-list* , *parameter-declaration*
- parameter-declaration*:
- declaration-specifiers* *declarator*
- declaration-specifiers* *abstract-declarator*<sub>opt</sub>
- identifier-list*:
- identifier*
- identifier-list* , *identifier*

### Semantics

- 2 Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.
- 3 A *full declarator* is a declarator that is not part of another declarator. The end of a full declarator is a sequence point. If the nested sequence of declarators in a full declarator

contains a variable length array type, the type specified by the full declarator is said to be *variably modified*.

- 4 In the following subclauses, consider a declaration

**T D1**

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

- 5 If, in the declaration “**T D1**”, **D1** has the form

*identifier*

then the type specified for *ident* is *T*.

- 6 If, in the declaration “**T D1**”, **D1** has the form

( **D** )

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

### Implementation limits

- 7 As discussed in 5.2.4.1, an implementation may limit the number of pointer, array, and function declarators that modify an arithmetic, structure, union, or incomplete type, either directly or via one or more **typedefs**.

**Forward references:** array declarators (6.7.5.2), type definitions (6.7.7).

### 6.7.5.1 Pointer declarators

#### Semantics

- 1 If, in the declaration “**T D1**”, **D1** has the form

*\* type-qualifier-list*<sub>opt</sub> **D**

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

- 2 For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types.
- 3 **EXAMPLE** The following pair of declarations demonstrates the difference between a “variable pointer to a constant value” and a “constant pointer to a variable value”.

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

The contents of any object pointed to by `ptr_to_constant` shall not be modified through that pointer,

but `ptr_to_constant` itself may be changed to point to another object. Similarly, the contents of the `int` pointed to by `constant_ptr` may be modified, but `constant_ptr` itself shall always point to the same location.

- 4 The declaration of the constant pointer `constant_ptr` may be clarified by including a definition for the type “pointer to `int`”.

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

declares `constant_ptr` as an object that has type “const-qualified pointer to `int`”.

### 6.7.5.2 Array declarators

#### Constraints

- 1 In addition to optional type qualifiers and the keyword `static`, the [ and ] may delimit an expression or `*`. If they delimit an expression (which specifies the size of an array), the expression shall have an integer type. If the expression is a constant expression, it shall have a value greater than zero. The element type shall not be an incomplete or function type. The optional type qualifiers and the keyword `static` shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.
- 2 Only an ordinary identifier (as defined in 6.2.3) with both block scope or function prototype scope and no linkage shall have a variably modified type. If an identifier is declared to be an object with static storage duration, it shall not have a variable length array type.

#### Semantics

- 3 If, in the declaration “`T D1`”, `D1` has one of the forms:

```
D[ type-qualifier-listopt assignment-expressionopt ]
D[ static type-qualifier-listopt assignment-expression ]
D[ type-qualifier-list static assignment-expression ]
D[ type-qualifier-listopt * ]
```

and the type specified for *ident* in the declaration “`T D`” is “*derived-declarator-type-list T*”, then the type specified for *ident* is “*derived-declarator-type-list array of T*”.<sup>121)</sup> (See 6.7.5.3 for the meaning of the optional type qualifiers and the keyword `static`.)

- 4 If the size is not present, the array type is an incomplete type. If the size is `*` instead of being an expression, the array type is a variable length array type of unspecified size, which can only be used in declarations with function prototype scope;<sup>122)</sup> such arrays are nonetheless complete types. If the size is an integer constant expression and the element

121) When several “array of” specifications are adjacent, a multidimensional array is declared.

122) Thus, `*` can be used only in function declarations that are not definitions (see 6.7.5.3).

type has a known constant size, the array type is not a variable length array type; otherwise, the array type is a variable length array type.

- 5 If the size is an expression that is not an integer constant expression: if it occurs in a declaration at function prototype scope, it is treated as if it were replaced by `*`; otherwise, each time it is evaluated it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where a size expression is part of the operand of a `sizeof` operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated.
- 6 For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers shall have the same constant value. If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values.

7 EXAMPLE 1

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

declares an array of `float` numbers and an array of pointers to `float` numbers.

8 EXAMPLE 2 Note the distinction between the declarations

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

The first declares `x` to be a pointer to `int`; the second declares `y` to be an array of `int` of unspecified size (an incomplete type), the storage for which is defined elsewhere.

9 EXAMPLE 3 The following declarations demonstrate the compatibility rules for variably modified types.

```
extern int n;
extern int m;
void fcompat(void)
{
    int a[n][6][m];
    int (*p)[4][n+1];
    int c[n][n][6][m];
    int (*r)[n][n][n+1];
    p = a;          // invalid: not compatible because 4 != 6
    r = c;          // compatible, but defined behavior only if
                    // n == 6 and m == n+1
}
```

- 10 EXAMPLE 4 All declarations of variably modified (VM) types have to be at either block scope or function prototype scope. Array objects declared with the **static** or **extern** storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the **static** storage-class specifier can have a VM type (that is, a pointer to a VLA type). Finally, all identifiers declared with a VM type have to be ordinary identifiers and cannot, therefore, be members of structures or unions.

```

extern int n;
int A[n]; // invalid: file scope VLA
extern int (*p2)[n]; // invalid: file scope VM
int B[100]; // valid: file scope but not VM

void fvla(int m, int C[m][m]); // valid: VLA with prototype scope

void fvla(int m, int C[m][m]) // valid: adjusted to auto pointer to VLA
{
    typedef int VLA[m][m]; // valid: block scope typedef VLA

    struct tag {
        int (*y)[n]; // invalid: y not ordinary identifier
        int z[n]; // invalid: z not ordinary identifier
    };
    int D[m]; // valid: auto VLA
    static int E[m]; // invalid: static block scope VLA
    extern int F[m]; // invalid: F has linkage and is VLA
    int (*s)[m]; // valid: auto pointer to VLA
    extern int (*r)[m]; // invalid: r has linkage and points to VLA
    static int (*q)[m] = &B; // valid: q is a static block pointer to VLA
}

```

**Forward references:** function declarators (6.7.5.3), function definitions (6.9.1), initialization (6.7.8).

### 6.7.5.3 Function declarators (including prototypes)

#### Constraints

- 1 A function declarator shall not specify a return type that is a function type or an array type.
- 2 The only storage-class specifier that shall occur in a parameter declaration is **register**.
- 3 An identifier list in a function declarator that is not part of a definition of that function shall be empty.
- 4 After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.

#### Semantics

- 5 If, in the declaration “**T D1**”, **D1** has the form

**D**( *parameter-type-list* )

or

**D**( *identifier-list<sub>opt</sub>* )

and the type specified for *ident* in the declaration “**T D**” is “*derived-declarator-type-list T*”, then the type specified for *ident* is “*derived-declarator-type-list function returning T*”.

- 6 A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function.
- 7 A declaration of a parameter as “array of *type*” shall be adjusted to “qualified pointer to *type*”, where the type qualifiers (if any) are those specified within the [ and ] of the array type derivation. If the keyword **static** also appears within the [ and ] of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.
- 8 A declaration of a parameter as “function returning *type*” shall be adjusted to “pointer to function returning *type*”, as in 6.3.2.1.
- 9 If the list terminates with an ellipsis (*, . . .*), no information about the number or types of the parameters after the comma is supplied.<sup>123)</sup>
- 10 The special case of an unnamed parameter of type **void** as the only item in the list specifies that the function has no parameters.
- 11 If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.
- 12 If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the [ \* ] notation in their sequences of declarator specifiers to specify variable length array types.
- 13 The storage-class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition.
- 14 An identifier list declares only the identifiers of the parameters of the function. An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.<sup>124)</sup>
- 15 For two function types to be compatible, both shall specify compatible return types.<sup>125)</sup>

---

123) The macros defined in the `<stdarg.h>` header (7.15) may be used to access arguments that correspond to the ellipsis.

124) See “future language directions” (6.11.6).

125) If both function types are “old style”, parameter types are not compared.

Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions. If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier. (In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.)

16 EXAMPLE 1 The declaration

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

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

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

18 EXAMPLE 2 The declaration

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

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

19 EXAMPLE 3 The declaration

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

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

- 20 EXAMPLE 4 The following prototype has a variably modified parameter.

```

void addscalar(int n, int m,
               double a[n][n*m+300], double x);

int main()
{
    double b[4][308];
    addscalar(4, 2, b, 2.17);
    return 0;
}

void addscalar(int n, int m,
               double a[n][n*m+300], double x)
{
    for (int i = 0; i < n; i++)
        for (int j = 0, k = n*m+300; j < k; j++)
            // a is a pointer to a VLA with n*m+300 elements
            a[i][j] += x;
}

```

- 21 EXAMPLE 5 The following are all compatible function prototype declarators.

```

double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);

```

as are:

```

void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);

```

(Note that the last declaration also specifies that the argument corresponding to **a** in any call to **f** must be a non-null pointer to the first of at least three arrays of 5 doubles, which the others do not.)

**Forward references:** function definitions (6.9.1), type names (6.7.6).

## 6.7.6 Type names

### Syntax

- 1 *type-name*:
- specifier-qualifier-list abstract-declarator<sub>opt</sub>*
- abstract-declarator*:
- pointer*
- pointer<sub>opt</sub> direct-abstract-declarator*
- direct-abstract-declarator*:
- ( *abstract-declarator* )
- direct-abstract-declarator<sub>opt</sub> [ assignment-expression<sub>opt</sub> ]*
- direct-abstract-declarator<sub>opt</sub> [ \* ]*
- direct-abstract-declarator<sub>opt</sub> ( parameter-type-list<sub>opt</sub> )*

### Semantics

- 2 In several contexts, it is necessary to specify a type. This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the identifier.<sup>126)</sup>
- 3 EXAMPLE The constructions
- (a) **int**
  - (b) **int \***
  - (c) **int \*[3]**
  - (d) **int (\*)[3]**
  - (e) **int (\*)[\*]**
  - (f) **int \*()**
  - (g) **int \*(void)**
  - (h) **int (\*const [ ])(unsigned int, ...)**

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

---

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

## 6.7.7 Type definitions

### Syntax

1           *typedef-name:*  
              *identifier*

### Constraints

2   If a typedef name specifies a variably modified type then it shall have block scope.

### Semantics

3   In a declaration whose storage-class specifier is **typedef**, each declarator defines an identifier to be a typedef name that denotes the type specified for the identifier in the way described in 6.7.5. Any array size expressions associated with variable length array declarators are evaluated each time the declaration of the typedef name is reached in the order of execution. A **typedef** declaration does not introduce a new type, only a synonym for the type so specified. That is, in the following declarations:

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

**type\_ident** is defined as a typedef name with the type specified by the declaration specifiers in **T** (known as *T*), and the identifier in **D** has the type “*derived-declarator-type-list T*” where the *derived-declarator-type-list* is specified by the declarators of **D**. A typedef name shares the same name space as other identifiers declared in ordinary declarators.

4   EXAMPLE 1   After

```
typedef int MILES, KCLICKSP();
typedef struct { double hi, lo; } range;
```

the constructions

```
MILES distance;
extern KCLICKSP *metricp;
range x;
range z, *zp;
```

are all valid declarations. The type of **distance** is **int**, that of **metricp** is “pointer to function with no parameter specification returning **int**”, and that of **x** and **z** is the specified structure; **zp** is a pointer to such a structure. The object **distance** has a type compatible with any other **int** object.

5   EXAMPLE 2   After the declarations

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type **t1** and the type pointed to by **tp1** are compatible. Type **t1** is also compatible with type **struct s1**, but not compatible with the types **struct s2**, **t2**, the type pointed to by **tp2**, or **int**.

- 6 EXAMPLE 3 The following obscure constructions

```
typedef signed int t;
typedef int plain;
struct tag {
    unsigned t:4;
    const t:5;
    plain r:5;
};
```

declare a typedef name **t** with type **signed int**, a typedef name **plain** with type **int**, and a structure with three bit-field members, one named **t** that contains values in the range [0, 15], an unnamed const-qualified bit-field which (if it could be accessed) would contain values in either the range [−15, +15] or [−16, +15], and one named **r** that contains values in one of the ranges [0, 31], [−15, +15], or [−16, +15]. (The choice of range is implementation-defined.) The first two bit-field declarations differ in that **unsigned** is a type specifier (which forces **t** to be the name of a structure member), while **const** is a type qualifier (which modifies **t** which is still visible as a typedef name). If these declarations are followed in an inner scope by

```
t f(t (t));
long t;
```

then a function **f** is declared with type “function returning **signed int** with one unnamed parameter with type pointer to function returning **signed int** with one unnamed parameter with type **signed int**”, and an identifier **t** with type **long int**.

- 7 EXAMPLE 4 On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the **signal** function specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

- 8 EXAMPLE 5 If a typedef name denotes a variable length array type, the length of the array is fixed at the time the typedef name is defined, not each time it is used:

```
void copyt(int n)
{
    typedef int B[n]; // B is n ints, n evaluated now
    n += 1;
    B a; // a is n ints, n without += 1
    int b[n]; // a and b are different sizes
    for (int i = 1; i < n; i++)
        a[i-1] = b[i];
}
```

## 6.7.8 Initialization

### Syntax

- 1        *initializer*:
- assignment-expression*  
           { *initializer-list* }  
           { *initializer-list* , }
- initializer-list*:
- designation<sub>opt</sub> initializer*  
           *initializer-list* , *designation<sub>opt</sub> initializer*
- designation*:
- designator-list* =
- designator-list*:
- designator*  
           *designator-list designator*
- designator*:
- [ *constant-expression* ]  
           . *identifier*

### Constraints

- 2        No initializer shall attempt to provide a value for an object not contained within the entity being initialized.
- 3        The type of the entity to be initialized shall be an array of unknown size or an object type that is not a variable length array type.
- 4        All the expressions in an initializer for an object that has static storage duration shall be constant expressions or string literals.
- 5        If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier.
- 6        If a designator has the form
- [ *constant-expression* ]
- then the current object (defined below) shall have array type and the expression shall be an integer constant expression. If the array is of unknown size, any nonnegative value is valid.
- 7        If a designator has the form
- . *identifier*
- then the current object (defined below) shall have structure or union type and the identifier shall be the name of a member of that type.

## Semantics

- 8 An initializer specifies the initial value stored in an object.
- 9 Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. Unnamed members of structure objects have indeterminate value even after initialization.
- 10 If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate. If an object that has static storage duration is not initialized explicitly, then:
  - if it has pointer type, it is initialized to a null pointer;
  - if it has arithmetic type, it is initialized to (positive or unsigned) zero;
  - if it is an aggregate, every member is initialized (recursively) according to these rules;
  - if it is a union, the first named member is initialized (recursively) according to these rules.
- 11 The initializer for a scalar shall be a single expression, optionally enclosed in braces. The initial value of the object is that of the expression (after conversion); the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type.
- 12 The rest of this subclause deals with initializers for objects that have aggregate or union type.
- 13 The initializer for a structure or union object that has automatic storage duration shall be either an initializer list as described below, or a single expression that has compatible structure or union type. In the latter case, the initial value of the object, including unnamed members, is that of the expression.
- 14 An array of character type may be initialized by a character string literal, optionally enclosed in braces. Successive characters of the character string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array.
- 15 An array with element type compatible with `wchar_t` may be initialized by a wide string literal, optionally enclosed in braces. Successive wide characters of the wide string literal (including the terminating null wide character if there is room or if the array is of unknown size) initialize the elements of the array.
- 16 Otherwise, the initializer for an object that has aggregate or union type shall be a brace-enclosed list of initializers for the elements or named members.
- 17 Each brace-enclosed initializer list has an associated *current object*. When no designations are present, subobjects of the current object are initialized in order according to the type of the current object: array elements in increasing subscript order, structure

members in declaration order, and the first named member of a union.<sup>127)</sup> In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator. Initialization then continues forward in order, beginning with the next subobject after that described by the designator.<sup>128)</sup>

- 18 Each designator list begins its description with the current object associated with the closest surrounding brace pair. Each item in the designator list (in order) specifies a particular member of its current object and changes the current object for the next designator (if any) to be that member.<sup>129)</sup> The current object that results at the end of the designator list is the subobject to be initialized by the following initializer.
- 19 The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;<sup>130)</sup> all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration.
- 20 If the aggregate or union contains elements or members that are aggregates or unions, these rules apply recursively to the subaggregates or contained unions. If the initializer of a subaggregate or contained union begins with a left brace, the initializers enclosed by that brace and its matching right brace initialize the elements or members of the subaggregate or the contained union. Otherwise, only enough initializers from the list are taken to account for the elements or members of the subaggregate or the first member of the contained union; any remaining initializers are left to initialize the next element or member of the aggregate of which the current subaggregate or contained union is a part.
- 21 If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration.
- 22 If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. At the end of its initializer list, the array no longer has incomplete type.

---

127) If the initializer list for a subaggregate or contained union does not begin with a left brace, its subobjects are initialized as usual, but the subaggregate or contained union does not become the current object: current objects are associated only with brace-enclosed initializer lists.

128) After a union member is initialized, the next object is not the next member of the union; instead, it is the next subobject of an object containing the union.

129) Thus, a designator can only specify a strict subobject of the aggregate or union that is associated with the surrounding brace pair. Note, too, that each separate designator list is independent.

130) Any initializer for the subobject which is overridden and so not used to initialize that subobject might not be evaluated at all.

23 The order in which any side effects occur among the initialization list expressions is unspecified.<sup>131)</sup>

24 EXAMPLE 1 Provided that `<complex.h>` has been `#included`, the declarations

```
int i = 3.5;
double complex c = 5 + 3 * I;
```

define and initialize `i` with the value 3 and `c` with the value  $5.0 + i3.0$ .

25 EXAMPLE 2 The declaration

```
int x[] = { 1, 3, 5 };
```

defines and initializes `x` as a one-dimensional array object that has three elements, as no size was specified and there are three initializers.

26 EXAMPLE 3 The declaration

```
int y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

is a definition with a fully bracketed initialization: 1, 3, and 5 initialize the first row of `y` (the array object `y[0]`), namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early, so `y[3]` is initialized with zeros. Precisely the same effect could have been achieved by

```
int y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y[0]` does not begin with a left brace, so three items from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`.

27 EXAMPLE 4 The declaration

```
int z[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `z` as specified and initializes the rest with zeros.

28 EXAMPLE 5 The declaration

```
struct { int a[3], b; } w[] = { { 1 }, 2 };
```

is a definition with an inconsistently bracketed initialization. It defines an array with two element structures: `w[0].a[0]` is 1 and `w[1].a[0]` is 2; all the other elements are zero.

---

131) In particular, the evaluation order need not be the same as the order of subobject initialization.

## 29 EXAMPLE 6 The declaration

```

short q[4][3][2] = {
    { 1 },
    { 2, 3 },
    { 4, 5, 6 }
};

```

contains an incompletely but consistently bracketed initialization. It defines a three-dimensional array object: `q[0][0][0]` is 1, `q[1][0][0]` is 2, `q[1][0][1]` is 3, and 4, 5, and 6 initialize `q[2][0][0]`, `q[2][0][1]`, and `q[2][1][0]`, respectively; all the rest are zero. The initializer for `q[0][0]` does not begin with a left brace, so up to six items from the current list may be used. There is only one, so the values for the remaining five elements are initialized with zero. Likewise, the initializers for `q[1][0]` and `q[2][0]` do not begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates. If there had been more than six items in any of the lists, a diagnostic message would have been issued. The same initialization result could have been achieved by:

```

short q[4][3][2] = {
    1, 0, 0, 0, 0, 0,
    2, 3, 0, 0, 0, 0,
    4, 5, 6
};

```

or by:

```

short q[4][3][2] = {
    {
        { 1 },
    },
    {
        { 2, 3 },
    },
    {
        { 4, 5 },
        { 6 },
    }
};

```

in a fully bracketed form.

- 30 Note that the fully bracketed and minimally bracketed forms of initialization are, in general, less likely to cause confusion.
- 31 EXAMPLE 7 One form of initialization that completes array types involves typedef names. Given the declaration

```

typedef int A[]; // OK - declared with block scope

```

the declaration

```

A a = { 1, 2 }, b = { 3, 4, 5 };

```

is identical to

```

int a[] = { 1, 2 }, b[] = { 3, 4, 5 };

```

due to the rules for incomplete types.

32 EXAMPLE 8 The declaration

```
char s[] = "abc", t[3] = "abc";
```

defines “plain” **char** array objects **s** and **t** whose elements are initialized with character string literals. This declaration is identical to

```
char s[] = { 'a', 'b', 'c', '\0' },
t[] = { 'a', 'b', 'c' };
```

The contents of the arrays are modifiable. On the other hand, the declaration

```
char *p = "abc";
```

defines **p** with type “pointer to **char**” and initializes it to point to an object with type “array of **char**” with length 4 whose elements are initialized with a character string literal. If an attempt is made to use **p** to modify the contents of the array, the behavior is undefined.

33 EXAMPLE 9 Arrays can be initialized to correspond to the elements of an enumeration by using designators:

```
enum { member_one, member_two };
const char *nm[] = {
    [member_two] = "member two",
    [member_one] = "member one",
};
```

34 EXAMPLE 10 Structure members can be initialized to nonzero values without depending on their order:

```
div_t answer = { .quot = 2, .rem = -1 };
```

35 EXAMPLE 11 Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood:

```
struct { int a[3], b; } w[] =
{ [0].a = {1}, [1].a[0] = 2 };
```

36 EXAMPLE 12 Space can be “allocated” from both ends of an array by using a single designator:

```
int a[MAX] = {
    1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0
};
```

37 In the above, if **MAX** is greater than ten, there will be some zero-valued elements in the middle; if it is less than ten, some of the values provided by the first five initializers will be overridden by the second five.

38 EXAMPLE 13 Any member of a union can be initialized:

```
union { /* ... */ } u = { .any_member = 42 };
```

**Forward references:** common definitions `<stddef.h>` (7.17).

## 6.8 Statements and blocks

### Syntax

- 1 *statement:*  
     *labeled-statement*  
     *compound-statement*  
     *expression-statement*  
     *selection-statement*  
     *iteration-statement*  
     *jump-statement*

### Semantics

- 2 A *statement* specifies an action to be performed. Except as indicated, statements are executed in sequence.
- 3 A *block* allows a set of declarations and statements to be grouped into one syntactic unit. The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear.
- 4 A *full expression* is an expression that is not part of another expression or of a declarator. Each of the following is a full expression: an initializer; the expression in an expression statement; the controlling expression of a selection statement (**if** or **switch**); the controlling expression of a **while** or **do** statement; each of the (optional) expressions of a **for** statement; the (optional) expression in a **return** statement. The end of a full expression is a sequence point.

**Forward references:** expression and null statements (6.8.3), selection statements (6.8.4), iteration statements (6.8.5), the **return** statement (6.8.6.4).

### 6.8.1 Labeled statements

#### Syntax

- 1 *labeled-statement:*  
     *identifier* : *statement*  
     **case** *constant-expression* : *statement*  
     **default** : *statement*

#### Constraints

- 2 A **case** or **default** label shall appear only in a **switch** statement. Further constraints on such labels are discussed under the **switch** statement.

- 3 Label names shall be unique within a function.

### Semantics

- 4 Any statement may be preceded by a prefix that declares an identifier as a label name. Labels in themselves do not alter the flow of control, which continues unimpeded across them.

**Forward references:** the `goto` statement (6.8.6.1), the `switch` statement (6.8.4.2).

## 6.8.2 Compound statement

### Syntax

- 1 *compound-statement:*  
     { *block-item-list*<sub>opt</sub> }
- block-item-list:*  
     *block-item*  
     *block-item-list block-item*
- block-item:*  
     *declaration*  
     *statement*

### Semantics

- 2 A *compound statement* is a block.

## 6.8.3 Expression and null statements

### Syntax

- 1 *expression-statement:*  
     *expression*<sub>opt</sub> ;

### Semantics

- 2 The expression in an expression statement is evaluated as a void expression for its side effects.<sup>132)</sup>
- 3 A *null statement* (consisting of just a semicolon) performs no operations.
- 4 **EXAMPLE 1** If a function call is evaluated as an expression statement for its side effects only, the discarding of its value may be made explicit by converting the expression to a void expression by means of a cast:

```
int p(int);
/* ... */
(void)p(0);
```

---

<sup>132)</sup> Such as assignments, and function calls which have side effects.

- 5 EXAMPLE 2 In the program fragment

```
char *s;
/* ... */
while (*s++ != '\0')
    ;
```

a null statement is used to supply an empty loop body to the iteration statement.

- 6 EXAMPLE 3 A null statement may also be used to carry a label just before the closing } of a compound statement.

```
while (loop1) {
    /* ... */
    while (loop2) {
        /* ... */
        if (want_out)
            goto end_loop1;
        /* ... */
    }
    /* ... */
end_loop1: ;
}
```

**Forward references:** iteration statements (6.8.5).

## 6.8.4 Selection statements

### Syntax

- 1 *selection-statement:*
- ```
if ( expression ) statement
if ( expression ) statement else statement
switch ( expression ) statement
```

### Semantics

- 2 A selection statement selects among a set of statements depending on the value of a controlling expression.
- 3 A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.

#### 6.8.4.1 The **if** statement

##### Constraints

- 1 The controlling expression of an **if** statement shall have scalar type.

##### Semantics

- 2 In both forms, the first substatement is executed if the expression compares unequal to 0. In the **else** form, the second substatement is executed if the expression compares equal

to 0. If the first substatement is reached via a label, the second substatement is not executed.

- 3 An **else** is associated with the lexically nearest preceding **if** that is allowed by the syntax.

#### 6.8.4.2 The **switch** statement

##### Constraints

- 1 The controlling expression of a **switch** statement shall have integer type.
- 2 If a **switch** statement has an associated **case** or **default** label within the scope of an identifier with a variably modified type, the entire **switch** statement shall be within the scope of that identifier.<sup>133)</sup>
- 3 The expression of each **case** label shall be an integer constant expression and no two of the **case** constant expressions in the same **switch** statement shall have the same value after conversion. There may be at most one **default** label in a **switch** statement. (Any enclosed **switch** statement may have a **default** label or **case** constant expressions with values that duplicate **case** constant expressions in the enclosing **switch** statement.)

##### Semantics

- 4 A **switch** statement causes control to jump to, into, or past the statement that is the *switch body*, depending on the value of a controlling expression, and on the presence of a **default** label and the values of any **case** labels on or in the switch body. A **case** or **default** label is accessible only within the closest enclosing **switch** statement.
- 5 The integer promotions are performed on the controlling expression. The constant expression in each **case** label is converted to the promoted type of the controlling expression. If a converted value matches that of the promoted controlling expression, control jumps to the statement following the matched **case** label. Otherwise, if there is a **default** label, control jumps to the labeled statement. If no converted **case** constant expression matches and there is no **default** label, no part of the switch body is executed.

##### Implementation limits

- 6 As discussed in 5.2.4.1, the implementation may limit the number of **case** values in a **switch** statement.

---

133) That is, the declaration either precedes the **switch** statement, or it follows the last **case** or **default** label associated with the **switch** that is in the block containing the declaration.

- 7 EXAMPLE In the artificial program fragment

```

switch (expr)
{
    int i = 4;
    f(i);
case 0:
    i = 17;
    /* falls through into default code */
default:
    printf("%d\n", i);
}

```

the object whose identifier is `i` exists with automatic storage duration (within the block) but is never initialized, and thus if the controlling expression has a nonzero value, the call to the `printf` function will access an indeterminate value. Similarly, the call to the function `f` cannot be reached.

## 6.8.5 Iteration statements

### Syntax

- 1 *iteration-statement*:
- ```

while ( expression ) statement
do statement while ( expression ) ;
for ( expressionopt ; expressionopt ; expressionopt ) statement
for ( declaration expressionopt ; expressionopt ) statement

```

### Constraints

- 2 The controlling expression of an iteration statement shall have scalar type.
- 3 The declaration part of a `for` statement shall only declare identifiers for objects having storage class `auto` or `register`.

### Semantics

- 4 An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0.
- 5 An iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. The loop body is also a block whose scope is a strict subset of the scope of the iteration statement.

### 6.8.5.1 The **while** statement

- 1 The evaluation of the controlling expression takes place before each execution of the loop body.

### 6.8.5.2 The **do** statement

- 1 The evaluation of the controlling expression takes place after each execution of the loop body.

### 6.8.5.3 The **for** statement

- 1 The statement

**for** ( *clause-1* ; *expression-2* ; *expression-3* ) *statement*

behaves as follows: The expression *expression-2* is the controlling expression that is evaluated before each execution of the loop body. The expression *expression-3* is evaluated as a void expression after each execution of the loop body. If *clause-1* is a declaration, the scope of any variables it declares is the remainder of the declaration and the entire loop, including the other two expressions; it is reached in the order of execution before the first evaluation of the controlling expression. If *clause-1* is an expression, it is evaluated as a void expression before the first evaluation of the controlling expression.<sup>134)</sup>

- 2 Both *clause-1* and *expression-3* can be omitted. An omitted *expression-2* is replaced by a nonzero constant.

## 6.8.6 Jump statements

### Syntax

- 1 *jump-statement*:
  - goto** *identifier* ;
  - continue** ;
  - break** ;
  - return** *expression<sub>opt</sub>* ;

### Semantics

- 2 A jump statement causes an unconditional jump to another place.

---

<sup>134)</sup> Thus, *clause-1* specifies initialization for the loop, possibly declaring one or more variables for use in the loop; the controlling expression, *expression-2*, specifies an evaluation made before each iteration, such that execution of the loop continues until the expression compares equal to 0; and *expression-3* specifies an operation (such as incrementing) that is performed after each iteration.

### 6.8.6.1 The `goto` statement

#### Constraints

- 1 The identifier in a `goto` statement shall name a label located somewhere in the enclosing function. A `goto` statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier.

#### Semantics

- 2 A `goto` statement causes an unconditional jump to the statement prefixed by the named label in the enclosing function.
- 3 **EXAMPLE 1** It is sometimes convenient to jump into the middle of a complicated set of statements. The following outline presents one possible approach to a problem based on these three assumptions:
  1. The general initialization code accesses objects only visible to the current function.
  2. The general initialization code is too large to warrant duplication.
  3. The code to determine the next operation is at the head of the loop. (To allow it to be reached by `continue` statements, for example.)

```

/* ... */
goto first_time;
for (;;) {
    // determine next operation
    /* ... */
    if (need to reinitialize) {
        // reinitialize-only code
        /* ... */
    first_time:
        // general initialization code
        /* ... */
        continue;
    }
    // handle other operations
    /* ... */
}

```

- 4 EXAMPLE 2 A **goto** statement is not allowed to jump past any declarations of objects with variably modified types. A jump within the scope, however, is permitted.

```

goto lab3;           // invalid: going INTO scope of VLA.
{
    double a[n];
    a[j] = 4.4;
lab3:
    a[j] = 3.3;
    goto lab4;       // valid: going WITHIN scope of VLA.
    a[j] = 5.5;
lab4:
    a[j] = 6.6;
}
goto lab4;           // invalid: going INTO scope of VLA.

```

### 6.8.6.2 The **continue** statement

#### Constraints

- 1 A **continue** statement shall appear only in or as a loop body.

#### Semantics

- 2 A **continue** statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body. More precisely, in each of the statements

|                                     |                                      |                                   |
|-------------------------------------|--------------------------------------|-----------------------------------|
| <b>while</b> ( <i>/* ... */</i> ) { | <b>do</b> {                          | <b>for</b> ( <i>/* ... */</i> ) { |
| <i>/* ... */</i>                    | <i>/* ... */</i>                     | <i>/* ... */</i>                  |
| <b>continue</b> ;                   | <b>continue</b> ;                    | <b>continue</b> ;                 |
| <i>/* ... */</i>                    | <i>/* ... */</i>                     | <i>/* ... */</i>                  |
| <b>contin</b> : ;                   | <b>contin</b> : ;                    | <b>contin</b> : ;                 |
| }                                   | } <b>while</b> ( <i>/* ... */</i> ); | }                                 |

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto contin**;<sup>135)</sup>

### 6.8.6.3 The **break** statement

#### Constraints

- 1 A **break** statement shall appear only in or as a switch body or loop body.

#### Semantics

- 2 A **break** statement terminates execution of the smallest enclosing **switch** or iteration statement.

---

<sup>135)</sup> Following the **contin**: label is a null statement.

### 6.8.6.4 The **return** statement

#### Constraints

- 1 A **return** statement with an expression shall not appear in a function whose return type is **void**. A **return** statement without an expression shall only appear in a function whose return type is **void**.

#### Semantics

- 2 A **return** statement terminates execution of the current function and returns control to its caller. A function may have any number of **return** statements.
- 3 If a **return** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.<sup>136)</sup>
- 4 EXAMPLE In:

```

    struct s { double i; } f(void);
    union {
        struct {
            int f1;
            struct s f2;
        } u1;
        struct {
            struct s f3;
            int f4;
        } u2;
    } g;

    struct s f(void)
    {
        return g.u1.f2;
    }

    /* ... */
    g.u2.f3 = f();

```

there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

---

136) The **return** statement is not an assignment. The overlap restriction of subclause 6.5.16.1 does not apply to the case of function return.

## 6.9 External definitions

### Syntax

- 1            *translation-unit*:  
                   *external-declaration*  
                   *translation-unit external-declaration*
- external-declaration*:  
                   *function-definition*  
                   *declaration*

### Constraints

- 2    The storage-class specifiers **auto** and **register** shall not appear in the declaration specifiers in an external declaration.
- 3    There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a **sizeof** operator whose result is an integer constant), there shall be exactly one external definition for the identifier in the translation unit.

### Semantics

- 4    As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. These are described as “external” because they appear outside any function (and hence have file scope). As discussed in 6.7, a declaration that also causes storage to be reserved for an object or a function named by the identifier is a definition.
- 5    An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **sizeof** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; otherwise, there shall be no more than one.<sup>137)</sup>

---

137) Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

## 6.9.1 Function definitions

### Syntax

- 1 *function-definition:*  
       *declaration-specifiers declarator declaration-list<sub>opt</sub> compound-statement*
- declaration-list:*  
       *declaration*  
       *declaration-list declaration*

### Constraints

- 2 The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.<sup>138)</sup>
- 3 The return type of a function shall be **void** or an object type other than array type.
- 4 The storage-class specifier, if any, in the declaration specifiers shall be either **extern** or **static**.
- 5 If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **void**, in which case there shall not be an identifier. No declaration list shall follow.
- 6 If the declarator includes an identifier list, each declaration in the declaration list shall have at least one declarator, those declarators shall declare only identifiers from the identifier list, and every identifier in the identifier list shall be declared. An identifier declared as a typedef name shall not be redeclared as a parameter. The declarations in the declaration list shall contain no storage-class specifier other than **register** and no initializations.

---

138) The intent is that the type category in a function definition cannot be inherited from a typedef:

```

typedef int F(void);           // type F is "function with no parameters
                               //           returning int"
F f, g;                       // f and g both have type compatible with F
F f { /* ... */ }            // WRONG: syntax/constraint error
F g() { /* ... */ }          // WRONG: declares that g returns a function
int f(void) { /* ... */ }    // RIGHT: f has type compatible with F
int g() { /* ... */ }        // RIGHT: g has type compatible with F
F *e(void) { /* ... */ }     // e returns a pointer to a function
F *(e)(void) { /* ... */ }   // same: parentheses irrelevant
int (*fp)(void);             // fp points to a function that has type F
F *Fp;                        // Fp points to a function that has type F

```

## Semantics

- 7 The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. If the declarator includes a parameter type list, the list also specifies the types of all the parameters; such a declarator also serves as a function prototype for later calls to the same function in the same translation unit. If the declarator includes an identifier list,<sup>139)</sup> the types of the parameters shall be declared in a following declaration list. In either case, the type of each parameter is adjusted as described in 6.7.5.3 for a parameter type list; the resulting type shall be an object type.
- 8 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.
- 9 Each parameter has automatic storage duration. Its identifier is an lvalue, which is in effect declared at the head of the compound statement that constitutes the function body (and therefore cannot be redeclared in the function body except in an enclosed block). The layout of the storage for parameters is unspecified.
- 10 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment. (Array expressions and function designators as arguments were converted to pointers before the call.)
- 11 After all parameters have been assigned, the compound statement that constitutes the body of the function definition is executed.
- 12 If the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined.
- 13 EXAMPLE 1 In the following:

```
extern int max(int a, int b)
{
    return a > b ? a : b;
}
```

**extern** is the storage-class specifier and **int** is the type specifier; **max(int a, int b)** is the function declarator; and

```
{ return a > b ? a : b; }
```

is the function body. The following similar definition uses the identifier-list form for the parameter declarations:

---

139) See “future language directions” (6.11.7).

```
extern int max(a, b)
int a, b;
{
    return a > b ? a : b;
}
```

Here `int a, b;` is the declaration list for the parameters. The difference between these two definitions is that the first form acts as a prototype declaration that forces conversion of the arguments of subsequent calls to the function, whereas the second form does not.

- 14 EXAMPLE 2 To pass one function to another, one might say

```
int f(void);
/* ... */
g(f);
```

Then the definition of `g` might read

```
void g(int (*funcp)(void))
{
    /* ... */
    (*funcp)() /* or funcp() ... */
}
```

or, equivalently,

```
void g(int func(void))
{
    /* ... */
    func() /* or (*func)() ... */
}
```

## 6.9.2 External object definitions

### Semantics

- 1 If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier.
- 2 A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*. If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0.
- 3 If the declaration of an identifier for an object is a tentative definition and has internal linkage, the declared type shall not be an incomplete type.

## 4 EXAMPLE 1

```
int i1 = 1;           // definition, external linkage
static int i2 = 2;   // definition, internal linkage
extern int i3 = 3;   // definition, external linkage
int i4;              // tentative definition, external linkage
static int i5;       // tentative definition, internal linkage

int i1;              // valid tentative definition, refers to previous
int i2;              // 6.2.2 renders undefined, linkage disagreement
int i3;              // valid tentative definition, refers to previous
int i4;              // valid tentative definition, refers to previous
int i5;              // 6.2.2 renders undefined, linkage disagreement

extern int i1;       // refers to previous, whose linkage is external
extern int i2;       // refers to previous, whose linkage is internal
extern int i3;       // refers to previous, whose linkage is external
extern int i4;       // refers to previous, whose linkage is external
extern int i5;       // refers to previous, whose linkage is internal
```

## 5 EXAMPLE 2 If at the end of the translation unit containing

```
int i[];
```

the array `i` still has incomplete type, the implicit initializer causes it to have one element, which is set to zero on program startup.

## 6.10 Preprocessing directives

### Syntax

```

1      preprocessing-file:
        groupopt

        group:
        group-part
        group group-part

        group-part:
        if-section
        control-line
        text-line
        # non-directive

        if-section:
        if-group elif-groupsopt else-groupopt endif-line

        if-group:
        # if      constant-expression new-line groupopt
        # ifdef   identifier new-line groupopt
        # ifndef identifier new-line groupopt

        elif-groups:
        elif-group
        elif-groups elif-group

        elif-group:
        # elif   constant-expression new-line groupopt

        else-group:
        # else   new-line groupopt

        endif-line:
        # endif new-line

```

*control-line:*

```
# include pp-tokens new-line
# define identifier replacement-list new-line
# define identifier lparen identifier-listopt )
                    replacement-list new-line
# define identifier lparen ... ) replacement-list new-line
# define identifier lparen identifier-list , ... )
                    replacement-list new-line

# undef identifier new-line
# line pp-tokens new-line
# error pp-tokensopt new-line
# pragma pp-tokensopt new-line
# new-line
```

*text-line:*

*pp-tokens<sub>opt</sub> new-line*

*non-directive:*

*pp-tokens new-line*

*lparen:*

a ( character not immediately preceded by white-space

*replacement-list:*

*pp-tokens<sub>opt</sub>*

*pp-tokens:*

*preprocessing-token*  
*pp-tokens preprocessing-token*

*new-line:*

the new-line character

## Description

- 2 A *preprocessing directive* consists of a sequence of preprocessing tokens that begins with a # preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character, and is ended by the next new-line character.<sup>140)</sup> A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

---

140) Thus, preprocessing directives are commonly called “lines”. These “lines” have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the # character string literal creation operator in 6.10.3.2, for example).

- 3 A text line shall not begin with a # preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.
- 4 When in a group that is skipped (6.10.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

### Constraints

- 5 The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

### Semantics

- 6 The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.
- 7 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.
- 8 EXAMPLE In:

```
#define EMPTY
EMPTY # include <file.h>
```

the sequence of preprocessing tokens on the second line is *not* a preprocessing directive, because it does not begin with a # at the start of translation phase 4, even though it will do so after the macro **EMPTY** has been replaced.

## 6.10.1 Conditional inclusion

### Constraints

- 1 The expression that controls conditional inclusion shall be an integer constant expression except that: it shall not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below;<sup>141)</sup> and it may contain unary operator expressions of the form

**defined** *identifier*

or

**defined** ( *identifier* )

which evaluate to 1 if the identifier is currently defined as a macro name (that is, if it is

---

141) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, etc.

predefined or if it has been the subject of a **#define** preprocessing directive without an intervening **#undef** directive with the same subject identifier), 0 if it is not.

### Semantics

#### 2 Preprocessing directives of the forms

```
# if constant-expression new-line groupopt
# elif constant-expression new-line groupopt
```

check whether the controlling constant expression evaluates to nonzero.

- 3 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the **defined** unary operator), just as in normal text. If the token **defined** is generated as a result of this replacement process or use of the **defined** unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the **defined** unary operator have been performed, all remaining identifiers are replaced with the pp-number **0**, and then each preprocessing token is converted into a token. The resulting tokens compose the controlling constant expression which is evaluated according to the rules of 6.6. For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types **intmax\_t** and **uintmax\_t** defined in the header **<stdint.h>**.<sup>142)</sup> This includes interpreting character constants, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character constants matches the value obtained when an identical character constant occurs in an expression (other than within a **#if** or **#elif** directive) is implementation-defined.<sup>143)</sup> Also, whether a single-character character constant may have a negative value is implementation-defined.

#### 4 Preprocessing directives of the forms

```
# ifdef identifier new-line groupopt
# ifndef identifier new-line groupopt
```

check whether the identifier is or is not currently defined as a macro name. Their

142) Thus, on an implementation where **INT\_MAX** is **0x7FFF** and **UINT\_MAX** is **0xFFFF**, the constant **0x8000** is signed and positive within a **#if** expression even though it would be unsigned in translation phase 7.

143) Thus, the constant expression in the following **#if** directive and **if** statement is not guaranteed to evaluate to the same value in these two contexts.

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

conditions are equivalent to **#if defined** *identifier* and **#if !defined** *identifier* respectively.

- 5 Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed. If none of the conditions evaluates to true, and there is a **#else** directive, the group controlled by the **#else** is processed; lacking a **#else** directive, all the groups until the **#endif** are skipped.<sup>144)</sup>

**Forward references:** macro replacement (6.10.3), source file inclusion (6.10.2), largest integer types (7.18.1.5).

## 6.10.2 Source file inclusion

### Constraints

- 1 A **#include** directive shall identify a header or source file that can be processed by the implementation.

### Semantics

- 2 A preprocessing directive of the form

```
# include <h-char-sequence> new-line
```

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the < and > delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

- 3 A preprocessing directive of the form

```
# include "q-char-sequence" new-line
```

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the " delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

```
# include <h-char-sequence> new-line
```

---

144) As indicated by the syntax, a preprocessing token shall not follow a **#else** or **#endif** directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.

with the identical contained sequence (including > characters, if any) from the original directive.

- 4 A preprocessing directive of the form

```
# include pp-tokens new-line
```

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **include** in the directive are processed just as in normal text. (Each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens.) The directive resulting after all replacements shall match one of the two previous forms.<sup>145)</sup> The method by which a sequence of preprocessing tokens between a < and a > preprocessing token pair or a pair of " characters is combined into a single header name preprocessing token is implementation-defined.

- 5 The implementation shall provide unique mappings for sequences consisting of one or more letters or digits (as defined in 5.2.1) followed by a period (.) and a single letter. The first character shall be a letter. The implementation may ignore the distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.
- 6 A **#include** preprocessing directive may appear in a source file that has been read because of a **#include** directive in another file, up to an implementation-defined nesting limit (see 5.2.4.1).

- 7 EXAMPLE 1 The most common uses of **#include** preprocessing directives are as in the following:

```
#include <stdio.h>
#include "myprog.h"
```

- 8 EXAMPLE 2 This illustrates macro-replaced **#include** directives:

```
#if VERSION == 1
    #define INCFILE "vers1.h"
#elif VERSION == 2
    #define INCFILE "vers2.h" // and so on
#else
    #define INCFILE "versN.h"
#endif
#include INCFILE
```

**Forward references:** macro replacement (6.10.3).

---

145) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 5.1.1.2); thus, an expansion that results in two string literals is an invalid directive.

### 6.10.3 Macro replacement

#### Constraints

- 1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.
- 2 An identifier currently defined as an object-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. Likewise, an identifier currently defined as a function-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical.
- 3 There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.
- 4 If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the `...`). There shall exist a `)` preprocessing token that terminates the invocation.
- 5 The identifier `__VA_ARGS__` shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.
- 6 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

#### Semantics

- 7 The identifier immediately following the **define** is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.
- 8 If a `#` preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.
- 9 A preprocessing directive of the form

**# define** *identifier replacement-list new-line*

defines an *object-like macro* that causes each subsequent instance of the macro name<sup>146)</sup> to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.

- 10 A preprocessing directive of the form

```
# define identifier lparen identifier-listopt ) replacement-list new-line
# define identifier lparen ... ) replacement-list new-line
# define identifier lparen identifier-list , ... ) replacement-list new-line
```

defines a *function-like macro* with arguments, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the **#define** preprocessing directive. Each subsequent instance of the function-like macro name followed by a ( as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching ) preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

- 11 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives,<sup>147)</sup> the behavior is undefined.
- 12 If there is a ... in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the *variable arguments*. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the ...).

---

146) Since, by macro-replacement time, all character constants and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 5.1.1.2, translation phases), they are never scanned for macro names or parameters.

147) Despite the name, a non-directive is a preprocessing directive.

### 6.10.3.1 Argument substitution

- 1 After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a `#` or `##` preprocessing token or followed by a `##` preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens are available.
- 2 An identifier `__VA_ARGS__` that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it.

### 6.10.3.2 The `#` operator

#### Constraints

- 1 Each `#` preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

#### Semantics

- 2 If, in the replacement list, a parameter is immediately preceded by a `#` preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token composing the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a `\` character is inserted before each `"` and `\` character of a character constant or string literal (including the delimiting `"` characters), except that it is implementation-defined whether a `\` character is inserted before the `\` character beginning a universal character name. If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is `""`. The order of evaluation of `#` and `##` operators is unspecified.

### 6.10.3.3 The ## operator

#### Constraints

- 1 A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

#### Semantics

- 2 If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a *placemaker* preprocessing token instead.<sup>148)</sup>
- 3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemaker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemaker preprocessing token, and concatenation of a placemaker with a non-placemaker preprocessing token results in the non-placemaker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.
- 4 EXAMPLE In the following fragment:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); // equivalent to
                    // char p[] = "x ## y";
```

The expansion produces, at various stages:

```
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"
```

In other words, expanding **hash\_hash** produces a new token, consisting of two adjacent sharp signs, but this new token is not the ## operator.

---

<sup>148)</sup> Placemaker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

### 6.10.3.4 Rescanning and further replacement

- 1 After all parameters in the replacement list have been substituted and **#** and **##** processing has taken place, all placemaker preprocessing tokens are removed. Then, the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.
- 2 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.
- 3 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 6.10.9 below.

### 6.10.3.5 Scope of macro definitions

- 1 A macro definition lasts (independent of block structure) until a corresponding **#undef** directive is encountered or (if none is encountered) until the end of the preprocessing translation unit. Macro definitions have no significance after translation phase 4.

- 2 A preprocessing directive of the form

```
# undef identifier new-line
```

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

- 3 EXAMPLE 1 The simplest use of this facility is to define a “manifest constant”, as in

```
#define TABSIZE 100
int table[TABSIZE];
```

- 4 EXAMPLE 2 The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

- 5 EXAMPLE 3 To illustrate the rules for redefinition and reexamination, the sequence

```
#define x      3
#define f(a)   f(x * (a))
#undef x
#define x      2
#define g      f
#define z      z[0]
#define h      g(~)
#define m(a)   a(w)
#define w      0,1
#define t(a)   a
#define p()    int
#define q(x)   x
#define r(x,y) x ## y
#define str(x) # x

f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
    (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };
```

results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0]))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~ 5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", "" };
```

- 6 EXAMPLE 4 To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \
    x ## s, x ## t)

#define INCFILE(n) vers ## n
#define glue(a, b) a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW    "hello"
#define LOW        LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4') // this goes away
    == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "=" %d, x" "2" "=" %s", x1, x2);
fputs(
    "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n",
    s);
#include "vers2.h"      (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs(
    "strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n",
    s);
#include "vers2.h"      (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the # and ## tokens in the macro definition is optional.

- 7 EXAMPLE 5 To illustrate the rules for placemaker preprocessing tokens, the sequence

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,,),
           t(10,,), t(,11,), t(,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89,
           10, 11, 12, };
```

- 8 EXAMPLE 6 To demonstrate the redefinition rules, the following sequence is valid.

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FUNC_LIKE(a)  ( a )
#define FUNC_LIKE( a )( /* note the white space */ \
                        a /* other stuff on this line
                        */ )
```

But the following redefinitions are invalid:

```
#define OBJ_LIKE      (0)          // different token sequence
#define OBJ_LIKE      (1 - 1)     // different white space
#define FUNC_LIKE(b)  ( a )       // different parameter usage
#define FUNC_LIKE(b)  ( b )       // different parameter spelling
```

- 9 EXAMPLE 7 Finally, to show the variable argument list macro facilities:

```
#define debug(...)    fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                           printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```
fprintf(stderr, "Flag" );
fprintf(stderr, "X = %d\n", x );
puts( "The first, second, and third items." );
((x>y)?puts("x>y"):
    printf("x is %d but y is %d", x, y));
```

## 6.10.4 Line control

### Constraints

- 1 The string literal of a **#line** directive, if present, shall be a character string literal.

### Semantics

- 2 The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (5.1.1.2) while processing the source file to the current token.
- 3 A preprocessing directive of the form

**# line** *digit-sequence new-line*

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). The digit sequence shall not specify zero, nor a number greater than 2147483647.

- 4 A preprocessing directive of the form

**# line** *digit-sequence "s-char-sequence<sub>opt</sub>" new-line*

sets the presumed line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

- 5 A preprocessing directive of the form

**# line** *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after **line** on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.

## 6.10.5 Error directive

### Semantics

- 1 A preprocessing directive of the form

```
# error pp-tokensopt new-line
```

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

## 6.10.6 Pragma directive

### Semantics

- 1 A preprocessing directive of the form

```
# pragma pp-tokensopt new-line
```

where the preprocessing token **STDC** does not immediately follow **pragma** in the directive (prior to any macro replacement)<sup>149)</sup> causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such **pragma** that is not recognized by the implementation is ignored.

- 2 If the preprocessing token **STDC** does immediately follow **pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms<sup>150)</sup> whose meanings are described elsewhere:

```
#pragma STDC FP_CONTRACT on-off-switch
```

```
#pragma STDC FENV_ACCESS on-off-switch
```

```
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

*on-off-switch*: one of

```
ON OFF DEFAULT
```

**Forward references:** the **FP\_CONTRACT** pragma (7.12.2), the **FENV\_ACCESS** pragma (7.6.1), the **CX\_LIMITED\_RANGE** pragma (7.3.4).

---

149) An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where **STDC** immediately follows **pragma**). If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined; an implementation is permitted to behave as if it were the standard pragma, but is not required to.

150) See “future language directions” (6.11.8).

## 6.10.7 Null directive

### Semantics

- 1 A preprocessing directive of the form

```
# new-line
```

has no effect.

## 6.10.8 Predefined macro names

- 1 The following macro names<sup>151)</sup> shall be defined by the implementation:

\_\_**DATE**\_\_ The date of translation of the preprocessing translation unit: a character string literal of the form "**Mmm dd yyyy**", where the names of the months are the same as those generated by the **asctime** function, and the first character of **dd** is a space character if the value is less than 10. If the date of translation is not available, an implementation-defined valid date shall be supplied.

\_\_**FILE**\_\_ The presumed name of the current source file (a character string literal).<sup>152)</sup>

\_\_**LINE**\_\_ The presumed line number (within the current source file) of the current source line (an integer constant).<sup>152)</sup>

\_\_**STDC**\_\_ The integer constant **1**, intended to indicate a conforming implementation.

\_\_**STDC\_HOSTED**\_\_ The integer constant **1** if the implementation is a hosted implementation or the integer constant **0** if it is not.

\_\_**STDC\_VERSION**\_\_ The integer constant **199901L**.<sup>153)</sup>

\_\_**TIME**\_\_ The time of translation of the preprocessing translation unit: a character string literal of the form "**hh:mm:ss**" as in the time generated by the **asctime** function. If the time of translation is not available, an implementation-defined valid time shall be supplied.

- 2 The following macro names are conditionally defined by the implementation:

\_\_**STDC\_IEC\_559**\_\_ The integer constant **1**, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic).

---

151) See “future language directions” (6.11.9).

152) The presumed source file name and line number can be changed by the **#line** directive.

153) This macro was not specified in ISO/IEC 9899:1990 and was specified as **199409L** in ISO/IEC 9899/AMD1:1995. The intention is that this will remain an integer constant of type **long int** that is increased with each revision of this International Standard.

`__STDC_IEC_559_COMPLEX__` The integer constant `1`, intended to indicate adherence to the specifications in informative annex G (IEC 60559 compatible complex arithmetic).

`__STDC_ISO_10646__` An integer constant of the form `yyyymmL` (for example, `199712L`). If this symbol is defined, then every character in the Unicode required set, when stored in an object of type `wchar_t`, has the same value as the short identifier of that character. The *Unicode required set* consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month.

- 3 The values of the predefined macros (except for `__FILE__` and `__LINE__`) remain constant throughout the translation unit.
- 4 None of these macro names, nor the identifier `defined`, shall be the subject of a `#define` or a `#undef` preprocessing directive. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.
- 5 The implementation shall not predefine the macro `__cplusplus`, nor shall it define it in any standard header.

**Forward references:** the `asctime` function (7.23.3.1), standard headers (7.1.2).

### 6.10.9 Pragma operator

#### Semantics

- 1 A unary operator expression of the form:

```
_Pragma ( string-literal )
```

is processed as follows: The string literal is *destringized* by deleting the `L` prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence `\"` by a double-quote, and replacing each escape sequence `\\` by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

- 2 EXAMPLE A directive of the form:

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ( "listing on \"..\listing.dir\"" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING ( ..\listing.dir )
```

## 6.11 Future language directions

### 6.11.1 Floating types

- 1 Future standardization may include additional floating-point types, including those with greater range, precision, or both than **long double**.

### 6.11.2 Linkages of identifiers

- 1 Declaring an identifier with internal linkage at file scope without the **static** storage-class specifier is an obsolescent feature.

### 6.11.3 External names

- 1 Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

### 6.11.4 Character escape sequences

- 1 Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

### 6.11.5 Storage-class specifiers

- 1 The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature.

### 6.11.6 Function declarators

- 1 The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

### 6.11.7 Function definitions

- 1 The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.

### 6.11.8 Pragma directives

- 1 Pragmas whose first preprocessing token is **STDC** are reserved for future standardization.

### 6.11.9 Predefined macro names

- 1 Macro names beginning with **\_\_STDC\_\_** are reserved for future standardization.

## 7. Library

### 7.1 Introduction

#### 7.1.1 Definitions of terms

- 1 A *string* is a contiguous sequence of characters terminated by and including the first null character. The term *multibyte string* is sometimes used instead to emphasize special processing given to multibyte characters contained in the string or to avoid confusion with a wide string. A *pointer to a string* is a pointer to its initial (lowest addressed) character. The *length of a string* is the number of bytes preceding the null character and the *value of a string* is the sequence of the values of the contained characters, in order.
- 2 The *decimal-point character* is the character used by functions that convert floating-point numbers to or from character sequences to denote the beginning of the fractional part of such character sequences.<sup>154)</sup> It is represented in the text and examples by a period, but may be changed by the **setlocale** function.
- 3 A *null wide character* is a wide character with code value zero.
- 4 A *wide string* is a contiguous sequence of wide characters terminated by and including the first null wide character. A *pointer to a wide string* is a pointer to its initial (lowest addressed) wide character. The *length of a wide string* is the number of wide characters preceding the null wide character and the *value of a wide string* is the sequence of code values of the contained wide characters, in order.
- 5 A *shift sequence* is a contiguous sequence of bytes within a multibyte string that (potentially) causes a change in shift state (see 5.2.1.2). A shift sequence shall not have a corresponding wide character; it is instead taken to be an adjunct to an adjacent multibyte character.<sup>155)</sup>

**Forward references:** character handling (7.4), the **setlocale** function (7.11.1.1).

---

154) The functions that make use of the decimal-point character are the numeric conversion functions (7.20.1, 7.24.4.1) and the formatted input/output functions (7.19.6, 7.24.2).

155) For state-dependent encodings, the values for **MB\_CUR\_MAX** and **MB\_LEN\_MAX** shall thus be large enough to count all the bytes in any complete multibyte character plus at least one adjacent shift sequence of maximum length. Whether these counts provide for more than one shift sequence is the implementation's choice.











































































































































































































































































































































































































































































































































































































































































































































































