

INTERNATIONAL STANDARD

ISO/IEC
9899

First edition
1990-12-15

Programming languages — C

Langages de programmation — C



Reference number
ISO/IEC 9899 : 1990 (E)



Documents

Document Sharing Hub

Contents

1	Scope	1
2	Normative references	1
3	Definitions and conventions	2
4	Compliance	3
5	Environment	5
5.1	Conceptual models	5
5.1.1	Translation environment	5
5.1.2	Execution environments	6
5.2	Environmental considerations	10
5.2.1	Character sets	10
5.2.2	Character display semantics	12
5.2.3	Signals and interrupts	12
5.2.4	Environmental limits	12
6	Language	18
6.1	Lexical elements	18
6.1.1	Keywords	19
6.1.2	Identifiers	19
6.1.3	Constants	25
6.1.4	String literals	30
6.1.5	Operators	31
6.1.6	Punctuators	32
6.1.7	Header names	32
6.1.8	Preprocessing numbers	33
6.1.9	Comments	33
6.2	Conversions	34
6.2.1	Arithmetic operands	34
6.2.2	Other operands	36
6.3	Expressions	38
6.3.1	Primary expressions	39
6.3.2	Postfix operators	39
6.3.3	Unary operators	43
6.3.4	Cast operators	45
6.3.5	Multiplicative operators	46
6.3.6	Additive operators	46
6.3.7	Bitwise shift operators	48
6.3.8	Relational operators	48
6.3.9	Equality operators	49
6.3.10	Bitwise AND operator	50
6.3.11	Bitwise exclusive OR operator	50
6.3.12	Bitwise inclusive OR operator	50
6.3.13	Logical AND operator	51
6.3.14	Logical OR operator	51
6.3.15	Conditional operator	51

© ISO/IEC 1990

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

6.3.16	Assignment operators	53
6.3.17	Comma operator	54
6.4	Constant expressions	55
6.5	Declarations	57
6.5.1	Storage-class specifiers	58
6.5.2	Type specifiers	58
6.5.3	Type qualifiers	64
6.5.4	Declarators	65
6.5.5	Type names	69
6.5.6	Type definitions	70
6.5.7	Initialization	71
6.6	Statements	75
6.6.1	Labeled statements	75
6.6.2	Compound statement, or block	75
6.6.3	Expression and null statements	76
6.6.4	Selection statements	77
6.6.5	Iteration statements	78
6.6.6	Jump statements	79
6.7	External definitions	81
6.7.1	Function definitions	81
6.7.2	External object definitions	83
6.8	Preprocessing directives	85
6.8.1	Conditional inclusion	86
6.8.2	Source file inclusion	87
6.8.3	Macro replacement	89
6.8.4	Line control	93
6.8.5	Error directive	93
6.8.6	Pragma directive	93
6.8.7	Null directive	94
6.8.8	Predefined macro names	94
6.9	Future language directions	95
6.9.1	External names	95
6.9.2	Character escape sequences	95
6.9.3	Storage-class specifiers	95
6.9.4	Function declarators	95
6.9.5	Function definitions	95
6.9.6	Array parameters	95
7	Library	96
7.1	Introduction	96
7.1.1	Definitions of terms	96
7.1.2	Standard headers	96
7.1.3	Reserved identifiers	97
7.1.4	Errors <errno.h>	97
7.1.5	Limits <float.h> and <limits.h>	98
7.1.6	Common definitions <stddef.h>	98
7.1.7	Use of library functions	99
7.2	Diagnostics <assert.h>	101
7.2.1	Program diagnostics	101
7.3	Character handling <ctype.h>	102
7.3.1	Character testing functions	102
7.3.2	Character case mapping functions	104
7.4	Localization <locale.h>	106
7.4.1	Locale control	107
7.4.2	Numeric formatting convention inquiry	108

7.5	Mathematics <math.h>	111
7.5.1	Treatment of error conditions	111
7.5.2	Trigonometric functions	111
7.5.3	Hyperbolic functions	113
7.5.4	Exponential and logarithmic functions	114
7.5.5	Power functions	115
7.5.6	Nearest integer, absolute value, and remainder functions	116
7.6	Nonlocal jumps <setjmp.h>	118
7.6.1	Save calling environment	118
7.6.2	Restore calling environment	119
7.7	Signal handling <signal.h>	120
7.7.1	Specify signal handling	120
7.7.2	Send signal	121
7.8	Variable arguments <stdarg.h>	122
7.8.1	Variable argument list access macros	122
7.9	Input/output <stdio.h>	124
7.9.1	Introduction	124
7.9.2	Streams	125
7.9.3	Files	126
7.9.4	Operations on files	127
7.9.5	File access functions	128
7.9.6	Formatted input/output functions	131
7.9.7	Character input/output functions	141
7.9.8	Direct input/output functions	144
7.9.9	File positioning functions	145
7.9.10	Error-handling functions	147
7.10	General utilities <stdlib.h>	149
7.10.1	String conversion functions	149
7.10.2	Pseudo-random sequence generation functions	153
7.10.3	Memory management functions	154
7.10.4	Communication with the environment	155
7.10.5	Searching and sorting utilities	157
7.10.6	Integer arithmetic functions	158
7.10.7	Multibyte character functions	159
7.10.8	Multibyte string functions	161
7.11	String handling <string.h>	162
7.11.1	String function conventions	162
7.11.2	Copying functions	162
7.11.3	Concatenation functions	163
7.11.4	Comparison functions	164
7.11.5	Search functions	165
7.11.6	Miscellaneous functions	168
7.12	Date and time <time.h>	170
7.12.1	Components of time	170
7.12.2	Time manipulation functions	170
7.12.3	Time conversion functions	172
7.13	Future library directions	176
7.13.1	Errors <errno.h>	176
7.13.2	Character handling <ctype.h>	176
7.13.3	Localization <locale.h>	176
7.13.4	Mathematics <math.h>	176
7.13.5	Signal handling <signal.h>	176
7.13.6	Input/output <stdio.h>	176
7.13.7	General utilities <stdlib.h>	176
7.13.8	String handling <string.h>	176

Annexes

A	Bibliography	177
B	Language syntax summary	178
B.1	Lexical grammar	178
B.2	Phrase structure grammar	182
B.3	Preprocessing directives	187
C	Sequence points	189
D	Library summary	190
D.1	Errors <code><errno.h></code>	190
D.2	Common definitions <code><stddef.h></code>	190
D.3	Diagnostics <code><assert.h></code>	190
D.4	Character handling <code><ctype.h></code>	190
D.5	Localization <code><locale.h></code>	190
D.6	Mathematics <code><math.h></code>	191
D.7	Nonlocal jumps <code><setjmp.h></code>	191
D.8	Signal handling <code><signal.h></code>	191
D.9	Variable arguments <code><stdarg.h></code>	192
D.10	Input/output <code><stdio.h></code>	192
D.11	General utilities <code><stdlib.h></code>	194
D.12	String handling <code><string.h></code>	195
D.13	Date and time <code><time.h></code>	195
E	Implementation limits	196
F	Common warnings	198
G	Portability issues	199
G.1	Unspecified behavior	199
G.2	Undefined behavior	200
G.3	Implementation-defined behavior	204
G.4	Locale-specific behavior	207
G.5	Common extensions	208
	Index	210

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

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.

International Standard ISO/IEC 9899 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

Annexes A, B, C, D, E, F and G are for information only.

Introduction

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.

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 [6.9] or library features [7.13]) is discouraged.

This International Standard is divided into four major subdivisions:

- the introduction and preliminary elements;
- the characteristics of environments that translate and execute C programs;
- the language syntax, constraints, and semantics;
- the library facilities.

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. A set of annexes summarizes information contained in this International Standard. The introduction, the examples, the footnotes, the references, and the annexes are not part of this International Standard.

The language clause (clause 7) is derived from “The C Reference Manual” (see annex A).

The library clause (clause 8) is based on the 1984 *usr/group Standard* (see annex A).

This page intentionally left blank

Programming languages — C

1 Scope

This International Standard specifies the form and establishes the interpretation of programs written in the C programming language.¹ It specifies

- the representation of C programs;
- the syntax and constraints of the C language;
- the semantic rules for interpreting C programs;
- the representation of input data to be processed by C programs;
- the representation of output data produced by C programs;
- the restrictions and limits imposed by a conforming implementation of C.

This International Standard does not specify

- the mechanism by which C programs are transformed for use by a data-processing system;
- the mechanism by which C programs are invoked for use by a data-processing system;
- the mechanism by which input data are transformed for use by a C program;
- the mechanism by which output data are transformed after being produced by a C program;
- the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;
- all minimal requirements of a data-processing system that is capable of supporting a conforming implementation.

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 646:1983, *Information processing — ISO 7-bit coded character set for information interchange*.

ISO 4217:1987, *Codes for the representation of currencies and funds*.

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. It is accompanied by a Rationale document that explains many of the decisions of the Technical Committee that produced it.

3 Definitions and conventions

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.

For the purposes of this International Standard, the following definitions apply. Other terms are defined at their first appearance, indicated by *italic* type. 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 2382.

3.1 alignment: A requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address.

3.2 argument: An 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. Also known as “actual argument” or “actual parameter.”

3.3 bit: The unit of data storage in the execution environment large enough to hold an object that may have one of two values. It need not be possible to express the address of each individual bit of an object.

3.4 byte: The unit of data storage large enough to hold any member of the basic character set of the execution environment. It shall be possible to express the address of each individual byte of an object uniquely. 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.

3.5 character: A bit representation that fits in a byte. The representation of each member of the basic character set in both the source and execution environments shall fit in a byte.

3.6 constraints: Syntactic and semantic restrictions by which the exposition of language elements is to be interpreted.

3.7 diagnostic message: A message belonging to an implementation-defined subset of the implementation’s message output.

3.8 forward references: References to later subclauses of this International Standard that contain additional information relevant to this subclause.

3.9 implementation: A 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.

3.10 implementation-defined behavior: Behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.

3.11 implementation limits: Restrictions imposed upon programs by the implementation.

3.12 locale-specific behavior: Behavior that depends on local conventions of nationality, culture, and language that each implementation shall document.

3.13 multibyte character: A sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set.

3.14 object: A region of data storage in the execution environment, the contents of which can represent values. 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. When referenced, an object may be interpreted as having a particular type; see 6.2.2.1.

3.15 parameter: An 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. Also known as “formal argument” or “formal parameter.”

3.16 undefined behavior: Behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which this International Standard imposes no requirements. Permissible 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).

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.”

3.17 unspecified behavior: Behavior, for a correct program construct and correct data, for which this International Standard explicitly imposes no requirements.

Examples

1. An example of unspecified behavior is the order in which the arguments to a function are evaluated.
2. An example of undefined behavior is the behavior on integer overflow.
3. An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.
4. An example of locale-specific behavior is whether the **islower** function returns true for characters other than the 26 lowercase English letters.

Forward references: bitwise shift operators (6.3.7), expressions (6.3), function calls (6.3.2.2), the **islower** function (7.3.1.6), localization (7.4).

4 Compliance

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

The two forms of *conforming implementation* are hosted and freestanding. A *conforming hosted implementation* shall accept any strictly conforming program. A *conforming freestanding implementation* shall accept any strictly conforming program in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers **<float.h>**, **<limits.h>**, **<stdarg.h>**, and **<stddef.h>**. A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.²

A *conforming program* is one that is acceptable to a conforming implementation.³

² This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this International Standard.

³ Strictly conforming programs are intended to be maximally portable among conforming implementations. Conforming programs may depend upon nonportable features of a conforming implementation.

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

Forward references: limits `<float.h>` and `<limits.h>` (7.1.5), variable arguments `<stdarg.h>` (7.8), common definitions `<stddef.h>` (7.1.6).

5 Environment

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 the environment clause (clause 5), 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

A C program need not all be translated at the same time. The text of the program is kept in units called *source files* in this International Standard. A source file together with all the headers and source files included via the preprocessing directive **#include**, less any source lines skipped by any of the conditional inclusion preprocessing directives, 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: conditional inclusion (6.8.1), linkages of identifiers (6.1.2.2), source file inclusion (6.8.2).

5.1.1.2 Translation phases

The precedence among the syntax rules of translation is specified by the following phases.⁴

1. Physical source file characters are mapped 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 new-line character and an immediately preceding backslash character is deleted, splicing physical source lines to form logical source lines. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character.
3. The source file is decomposed into preprocessing tokens⁵ and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or 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 and macro invocations are expanded. A **#include** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.

⁴ Implementations must behave as if these separate phases occur, even though many are typically folded together in practice.

⁵ As described in 6.1, 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.

5. Each source character set member and escape sequence in character constants and string literals is converted to a member of the execution character set.
6. Adjacent character string literal tokens are concatenated and adjacent wide 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.
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: lexical elements (6.1), preprocessing directives (6.8), trigraph sequences (5.2.1.1).

5.1.1.3 Diagnostics

A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) for every translation unit that contains a violation of any syntax rule or constraint. Diagnostic messages need not be produced in other circumstances.⁶

5.1.2 Execution environments

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 in static storage 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: initialization (6.5.7).

5.1.2.1 Freestanding environment

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. There are otherwise no reserved external identifiers. Any library facilities available to a freestanding program are implementation-defined.

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

5.1.2.2 Hosted environment

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

5.1.2.2.1 Program startup

The function called at program startup is named **main**. The implementation declares no prototype for this function. It can be defined 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):

⁶ 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.

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

If they are defined, 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

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

5.1.2.2.3 Program termination

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. If the **main** function executes a return that specifies no value, the termination status returned to the host environment is undefined.

Forward references: definition of terms (7.1.1), the **exit** function (7.10.4.3).

5.1.2.3 Program execution

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

Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*, 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.

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).

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.

An instance of each object with automatic storage duration is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

The least requirements on a conforming implementation are:

- At sequence points, volatile objects are stable in the sense that previous evaluations are complete and subsequent evaluations have not yet occurred.
- 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 7.9.3. 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.

What constitutes an interactive device is implementation-defined.

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

Examples

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.

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.

2. In executing the fragment

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

the “integral 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 creating an overflow exception, the actual execution need only produce the same result, possibly omitting the promotions.

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**). Alternatively, an operation involving only **ints** or **floats** may be executed using double-precision operations if neither range nor precision is lost thereby.

4. 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 exception 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 an exception 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 overflows do not produce an exception and in which the results of overflows are reversible, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

5. 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: compound statement, or block (6.6.2), expressions (6.3), files (7.9.3), sequence points (6.3, 6.6), the **signal** function (7.7), type qualifiers (6.5.3).

5.2 Environmental considerations

5.2.1 Character sets

Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written, and the set interpreted in the execution environment. The values of the members of the execution character set are implementation-defined; any additional members beyond those required by this subclause are locale-specific.

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 literal.

Both the basic source and basic execution character sets shall have at least the following members: the 26 uppercase letters of the English alphabet

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

the 26 lowercase letters of the English alphabet

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

the 10 decimal digits

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

the following 29 graphic characters

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

the space character, and control characters representing horizontal tab, vertical tab, and form feed. 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 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 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.

Forward references: character constants (6.1.3.4), preprocessing directives (6.8), string literals (6.1.4), comments (6.1.9).

5.2.1.1 Trigraph sequences

All occurrences in a source file of the following sequences of three characters (called *trigraph sequences*⁷) are replaced with the corresponding single character.

⁷ The trigraph sequences enable the input of characters that are not defined in the Invariant Code Set as described in ISO 646:1983, which is a subset of the seven-bit ASCII code set.

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

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

Example

The following source line

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

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

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

5.2.1.2 Multibyte characters

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 single-byte characters defined in 5.2.1 shall be present.
- The presence, meaning, and representation of any additional members is locale-specific.
- A multibyte character may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other implementation-defined *shift states* when specific multibyte characters are encountered in the sequence. While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state. The interpretation for subsequent bytes in the sequence is a function of the current shift state.
- A byte with all bits zero shall be interpreted as a null character independent of shift state.
- A byte with all bits zero shall not occur in the second or subsequent bytes of a multibyte character.

For the source character set, the following shall hold:

- A comment, string literal, character constant, or header name shall begin and end in the initial shift state.
- A comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters.

5.2.2 Character display semantics

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

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. The active position shall not be changed.
- \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 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 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 is unspecified.

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

Forward references: the **fputc** function (7.9.7.3), the **isprint** function (7.3.1.7).

5.2.3 Signals and interrupts

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 comprise the executable representation of a function) on a per-invocation basis.

The functions in the standard library are not guaranteed to be reentrant and may modify objects with static storage duration.

5.2.4 Environmental limits

Both the translation and execution environments constrain the implementation of language translators and libraries. The following summarizes the environmental limits on a conforming implementation.

5.2.4.1 Translation limits

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:⁸

⁸ Implementations should avoid imposing fixed translation limits whenever possible.

- 15 nesting levels of compound statements, iteration control structures, and selection control structures
- 8 nesting levels of conditional inclusion
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration
- 31 nesting levels of parenthesized declarators within a full declarator
- 32 nesting levels of parenthesized expressions within a full expression
- 31 significant initial characters in an internal identifier or a macro name
- 6 significant initial characters in an external identifier
- 511 external identifiers in one translation unit
- 127 identifiers with block scope declared in one block
- 1024 macro identifiers simultaneously defined in one translation unit
- 31 parameters in one function definition
- 31 arguments in one function call
- 31 parameters in one macro definition
- 31 arguments in one macro invocation
- 509 characters in a logical source line
- 509 characters in a character string literal or wide string literal (after concatenation)
- 32767 bytes in an object (in a hosted environment only)
- 8 nesting levels for **#included** files
- 257 **case** labels for a **switch** statement (excluding those for any nested **switch** statements)
- 127 members in a single structure or union
- 127 enumeration constants in a single enumeration
- 15 levels of nested structure or union definitions in a single struct-declaration-list

5.2.4.2 Numerical limits

A conforming implementation shall document all the limits specified in this subclause, which shall be specified in the headers **<limits.h>** and **<float.h>**.

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

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 integral 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
- maximum value for an object of type **signed char**
SCHAR_MAX +127

- maximum value for an object of type **unsigned char**
UCHAR_MAX 255
- 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
- maximum value for an object of type **short int**
SHRT_MAX +32767
- maximum value for an object of type **unsigned short int**
USHRT_MAX 65535
- minimum value for an object of type **int**
INT_MIN -32767
- maximum value for an object of type **int**
INT_MAX +32767
- maximum value for an object of type **unsigned int**
UINT_MAX 65535
- minimum value for an object of type **long int**
LONG_MIN -2147483647
- maximum value for an object of type **long int**
LONG_MAX +2147483647
- maximum value for an object of type **unsigned long int**
ULONG_MAX 4294967295

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**.⁹

5.2.4.2.2 Characteristics of floating types <float.h>

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.¹⁰ The following parameters are used to define the model for each floating-point type:

⁹ See 6.1.2.5.

¹⁰ 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.

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

A normalized floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:

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

Of the values in the `<float.h>` header, **FLT_RADIX** shall be a constant expression suitable for use in `#if` preprocessing directives; all other values need not be constant expressions. All except **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_ROUNDS**.

The rounding mode for floating-point addition is characterized by the value of **FLT_ROUNDS**:

-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.

The values given in the following list shall be replaced by implementation-defined expressions that shall be equal or greater 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, 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, $\left\lfloor (p-1) \times \log_{10} b \right\rfloor + \begin{cases} 1 & \text{if } b \text{ is a power of } 10 \\ 0 & \text{otherwise} \end{cases}$

FLT_DIG 6

DBL_DIG 10

LDBL_DIG 10

— minimum negative integer such that **FLT_RADIX** raised to that power minus 1 is a normalized floating-point number, $e_{\min}$

FLT_MIN_EXP

DBL_MIN_EXP

LDBL_MIN_EXP

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

FLT_MIN_10_EXP -37

DBL_MIN_10_EXP -37

LDBL_MIN_10_EXP -37

- maximum integer such that **FLT_RADIX** raised to that power minus 1 is a representable finite floating-point number, $e_{\max}$

```
FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP
```

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

```

FLT_MAX_10_EXP      +37
DBL_MAX_10_EXP      +37
LDBL_MAX_10_EXP     +37

```

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

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

```

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

```

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

- the difference between 1 and the least value greater than 1 that is representable in the given floating point type, b^{1-p}

```

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

```

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

```

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

```

Examples

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`:

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

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

2. The following describes floating-point representations that also meet the requirements for single-precision and double-precision normalized numbers in ANSI/IEEE 754-1985,¹¹ and the appropriate values in a `<float.h>` header for types `float` and `double`:

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

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

FLT_RADIX	2
FLT_MANT_DIG	24
FLT_EPSILON	1.19209290E-07F
FLT_DIG	6
FLT_MIN_EXP	-125
FLT_MIN	1.17549435E-38F
FLT_MIN_10_EXP	-37
FLT_MAX_EXP	+128
FLT_MAX	3.40282347E+38F
FLT_MAX_10_EXP	+38
DBL_MANT_DIG	53
DBL_EPSILON	2.2204460492503131E-16
DBL_DIG	15
DBL_MIN_EXP	-1021
DBL_MIN	2.2250738585072014E-308
DBL_MIN_10_EXP	-307
DBL_MAX_EXP	+1024
DBL_MAX	1.7976931348623157E+308
DBL_MAX_10_EXP	+308

Forward references: conditional inclusion (6.8.1).

¹¹ 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.

6 Language

In the syntax notation used in the language clause (clause 6), 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

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

indicates an optional expression enclosed in braces.

6.1 Lexical elements

Syntax

token:

keyword
identifier
constant
string-literal
operator
punctuator

preprocessing-token:

header-name
identifier
pp-number
character-constant
string-literal
operator
punctuator
 each non-white-space character that cannot be one of the above

Constraints

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

Semantics

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*, *operators*, and *punctuators*. A *preprocessing token* is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: *header names*, *identifiers*, *preprocessing numbers*, *character constants*, *string literals*, *operators*, *punctuators*, and single non-white-space characters that do not lexically match the other preprocessing token categories. 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 6.8, 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.

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.

Examples

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.
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 (6.1.3.4), comments (6.1.9), expressions (6.3), floating constants (6.1.3.1), header names (6.1.7), macro replacement (6.8.3), postfix increment and decrement operators (6.3.2.4), prefix increment and decrement operators (6.3.3.1), preprocessing directives (6.8), preprocessing numbers (6.1.8), string literals (6.1.4).

6.1.1 Keywords

Syntax

keyword: one of

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Semantics

The above tokens (entirely in lowercase) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise.

6.1.2 Identifiers

Syntax

identifier:

nondigit
identifier nondigit
identifier digit

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
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------

Description

An identifier is a sequence of nondigit characters (including the underscore **_** and the lowercase and uppercase letters) and digits. The first character shall be a nondigit character.

Constraints

In translation phases 7 and 8, an identifier shall not consist of the same sequence of characters as a keyword.

Semantics

An identifier denotes an object, a function, or one of the following entities that will be described later: a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter. 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.

There is no specific limit on the maximum length of an identifier.

Implementation limits

The implementation shall treat at least the first 31 characters of an *internal name* (a macro name or an identifier that does not have external linkage) as significant. Corresponding lowercase and uppercase letters are different. The implementation may further restrict the significance of an *external name* (an identifier that has external linkage) to six characters and may ignore distinctions of alphabetical case for such names.¹² These limitations on identifiers are all implementation-defined.

Any identifiers that differ in a significant character are different identifiers. If two identifiers differ in a nonsignificant character, the behavior is undefined.

Forward references: linkages of identifiers (6.1.2.2), macro replacement (6.8.3).

6.1.2.1 Scopes of identifiers

An identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. 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.)

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). Label names shall be unique within a function.

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 **}** that closes 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 outer declaration of a lexically identical identifier exists in the same name space, it is hidden until the current scope terminates, after which it again becomes visible.

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

Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has scope that begins just after the completion of its declarator.

¹² See “future language directions” (6.9.1).

Forward references: compound statement, or block (6.6.2), declarations (6.5), enumeration specifiers (6.5.2.2), function calls (6.3.2.2), function declarators (including prototypes) (6.5.4.3), function definitions (6.7.1), the **goto** statement (6.6.6.1), labeled statements (6.6.1), name spaces of identifiers (6.1.2.3), scope of macro definitions (6.8.3.5), source file inclusion (6.8.2), tags (6.5.2.3), type specifiers (6.5.2).

6.1.2.2 Linkages of identifiers

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

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

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.¹³

If the declaration of an identifier for an object or a function contains the storage-class specifier **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage.

If the declaration of an identifier for a function has no storage-class specifier, its linkage is determined exactly as if it were declared with the storage-class specifier **extern**. If the declaration of an identifier for an object has file scope and no storage-class specifier, its linkage is external.

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**.

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

Forward references: compound statement, or block (6.6.2), declarations (6.5), expressions (6.3), external definitions (6.7).

6.1.2.3 Name spaces of identifiers

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 any¹⁴ 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);

¹³ A function declaration can contain the storage-class specifier **static** only if it is at file scope; see 6.5.1.

¹⁴ There is only one name space for tags even though three are possible.

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

Forward references: enumeration specifiers (6.5.2.2), labeled statements (6.6.1), structure and union specifiers (6.5.2.1), structure and union members (6.3.2.3), tags (6.5.2.3).

6.1.2.4 Storage durations of objects

An object has a *storage duration* that determines its lifetime. There are two storage durations: static and automatic.

An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **static** has *static storage duration*. For such an object, storage is reserved and its stored value is initialized only once, prior to program startup. The object exists and retains its last-stored value throughout the execution of the entire program.¹⁵

An object whose identifier is declared with no linkage and without the storage-class specifier **static** has *automatic storage duration*. Storage is guaranteed to be reserved for a new instance of such an object on each normal entry into the block with which it is associated, or on a jump from outside the block to a labeled statement in the block or in an enclosed block. If an initialization is specified for the value stored in the object, it is performed on each normal entry, but not if the block is entered by a jump to a labeled statement. Storage for the object is no longer guaranteed to be reserved when execution of the block ends in any way. (Entering an enclosed block suspends but does not end execution of the enclosing block. Calling a function suspends but does not end execution of the block containing the call.) The value of a pointer that referred to an object with automatic storage duration that is no longer guaranteed to be reserved is indeterminate.

Forward references: compound statement, or block (6.6.2), function calls (6.3.2.2), initialization (6.5.7).

6.1.2.5 Types

The meaning of a value stored in an object or returned by a function is determined by the *type* of the expression used to access it. (An identifier declared to be an object is the simplest such expression; the type is specified in the declaration of the identifier.) Types are partitioned into *object types* (types that describe objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes).

An object declared as type **char** is large enough to store any member of the basic execution character set. If a member of the required source character set enumerated in 5.2.1 is stored in a **char** object, its value is guaranteed to be positive. If other quantities are stored in a **char** object, the behavior is implementation-defined: the values are treated as either signed or nonnegative integers.

There are four *signed integer types*, designated as **signed char**, **short int**, **int**, and **long int**. (The signed integer and other types may be designated in several additional ways, as described in 6.5.2.)

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>**). In the list of signed integer types above, the range of values of each type is a subrange of the values of the next type in the list.

¹⁵ In the case of a volatile object, the last store may not be explicit in the program.

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 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.¹⁶ 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 unsigned integer type.

There are three *floating types*, designated as **float**, **double**, and **long double**. 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**.

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.

The three types **char**, **signed char**, and **unsigned char** are collectively called the *character types*.

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

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

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*.¹⁷ 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, each of which has an optionally specified name and possibly distinct type.
- A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type.
- A *function type* describes a function with specified return type. A function type is characterized by its return type and the number and types of its parameters. A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called “function returning *T*.” The construction of a function type from a return type is called “function type derivation.”
- A *pointer type* may be derived from a function type, 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.”

¹⁶ The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

¹⁷ Since object types do not include incomplete types, an array of incomplete type cannot be constructed.

These methods of constructing derived types can be applied recursively.

The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integral types*. The representations of integral types shall define values by use of a pure binary numeration system.¹⁸ The representations of floating types are unspecified.

Integral and floating types are collectively called *arithmetic types*. Arithmetic types and pointer types are collectively called *scalar types*. Array and structure types are collectively called *aggregate types*.¹⁹

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 6.5.2.3) 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.

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*.

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.

Any type so far mentioned is an *unqualified type*. Each unqualified type has three corresponding *qualified versions* of its type:²⁰ a *const-qualified* version, a *volatile-qualified* version, and a version having both qualifications. 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.¹⁶ A derived type is not qualified by the qualifiers (if any) of the type from which it is derived.

A pointer to **void** shall have the same representation and alignment requirements as a pointer to a character type. Similarly, pointers to qualified or unqualified versions of compatible types shall have the same representation and alignment requirements.¹⁶ Pointers to other types need not have the same representation or alignment requirements.

Examples

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.
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: character constants (6.1.3.4), compatible type and composite type (6.1.2.6), declarations (6.5), tags (6.5.2.3), type qualifiers (6.5.3).

¹⁸ 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*.)

¹⁹ Note that aggregate type does not include union type because an object with union type can only contain one member at a time.

²⁰ See 6.5.3 regarding qualified array and function types.

6.1.2.6 Compatible type and composite type

Two types have *compatible type* if their types are the same. Additional rules for determining whether two types are compatible are described in 6.5.2 for type specifiers, in 6.5.3 for type qualifiers, and in 6.5.4 for declarators.²¹ Moreover, two structure, union, or enumeration types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types; for two structures, the members shall be in the same order; for two structures or unions, the bit-fields shall have the same widths; for two enumerations, the members shall have the same values.

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

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 size, the composite type is an array of that size.
- 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.

For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type.

Example

Given the following two file scope declarations:

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

The resulting composite type for the function is:

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

Forward references: declarators (6.5.4), enumeration specifiers (6.5.2.2), structure and union specifiers (6.5.2.1), type definitions (6.5.6), type qualifiers (6.5.3), type specifiers (6.5.2).

6.1.3 Constants

Syntax

```
constant:
    floating-constant
    integer-constant
    enumeration-constant
    character-constant
```

Constraints

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

²¹ Two types need not be identical to be compatible.

Semantics

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

6.1.3.1 Floating constants

Syntax

floating-constant:
fractional-constant *exponent-part*_{opt} *floating-suffix*_{opt}
digit-sequence *exponent-part* *floating-suffix*_{opt}

fractional-constant:
*digit-sequence*_{opt} . *digit-sequence*
digit-sequence .

exponent-part:
e *sign*_{opt} *digit-sequence*
E *sign*_{opt} *digit-sequence*

sign: one of
+ -

digit-sequence:
digit
digit-sequence *digit*

floating-suffix: one of
f **l** **F** **L**

Description

A floating constant has a *significand part* that may be followed by an *exponent part* and a suffix that specifies its type. The components of the significand 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** or **E** followed by an exponent consisting of an optionally signed digit sequence. Either the whole-number part or the fraction part shall be present; either the period or the exponent part shall be present.

Semantics

The significand part is interpreted as a decimal rational number; the digit sequence in the exponent part is interpreted as a decimal integer. The exponent indicates the power of 10 by which the significand part is to be scaled. If the scaled value is in the range of representable values (for its type) 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.

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**.

6.1.3.2 Integer constants

Syntax

integer-constant:
decimal-constant *integer-suffix*_{opt}
octal-constant *integer-suffix*_{opt}
hexadecimal-constant *integer-suffix*_{opt}

decimal-constant:
 nonzero-digit
 decimal-constant digit

octal-constant:
 0
 octal-constant octal-digit

hexadecimal-constant:
 0*x* *hexadecimal-digit*
 0*X* *hexadecimal-digit*
 hexadecimal-constant hexadecimal-digit

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_{opt}
 long-suffix unsigned-suffix_{opt}

unsigned-suffix: one of
 u U

long-suffix: one of
 l L

Description

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.

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 0*x* or 0*X* 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

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.

The type of an integer constant is the first of the corresponding list in which its value can be represented. Unsuffix decimal: **int**, **long int**, **unsigned long int**; unsuffix octal or hexadecimal: **int**, **unsigned int**, **long int**, **unsigned long int**; suffixed by the letter u or U: **unsigned int**, **unsigned long int**; suffixed by the letter l or L: **long int**, **unsigned long int**; suffixed by both the letters u or U and l or L: **unsigned long int**.

6.1.3.3 Enumeration constants

Syntax

enumeration-constant:
identifier

Semantics

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

Forward references: enumeration specifiers (6.5.2.2).

6.1.3.4 Character constants

Syntax

character-constant:

' c-char-sequence'

L*' 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

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

An integer character constant is a sequence of one or more multibyte characters enclosed in single-quotes, as in '**x**' or '**ab**'. 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.

The single-quote ' , the double-quote " , the question-mark ? , the backslash \ , and arbitrary integral values, are representable according to the following table of escape sequences:

single-quote '	\'
double-quote "	\"
question-mark ?	\?
backslash \	\\
octal integer	\octal digits
hexadecimal integer	\xhexadecimal digits

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 \\.

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.

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.

Each octal or hexadecimal escape sequence is the longest sequence of characters that can constitute the escape sequence.

In addition, 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.²² If any other escape sequence is encountered, the behavior is undefined.²³

Constraints

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

An integer character constant has type **int**. The value of an integer character constant containing a single character that maps into a member of the basic execution character set 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, or containing a character or escape sequence not represented in the basic execution character set, 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**.

A wide character constant has type **wchar_t**, an integral type defined in the **<stddef.h>** header. The value of a wide character constant containing a single multibyte character that maps into a member of the extended execution character set is the *wide character* (code) 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.

²² The semantics of these characters were discussed in 5.2.2.

²³ See "future language directions" (6.9.2).

Examples

1. The construction `'\0'` is commonly used to represent the null character.
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 $+255$.
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. (The value of this single-character integer character constant is implementation-defined and violates the above constraint.) To specify an integer character constant containing the two characters whose values are `0x12` and `'3'`, the construction `'\0223'` may be used, since a hexadecimal escape sequence is terminated only by a nonhexadecimal character. (The value of this two-character integer character constant is implementation-defined also.)
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: characters and integers (6.2.1.1) common definitions `<stddef.h>` (7.1.6), the `mbtowc` function (7.10.7.2).

6.1.4 String literals

Syntax

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

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`.

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

In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character string literal tokens, or adjacent wide string literal tokens, are concatenated into a single multibyte character sequence. If a character string literal token is adjacent to a wide string literal token, the behavior is undefined.

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.²⁴ 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 sequence.

Identical string literals of either form need not be distinct. If the program attempts to modify a string literal of either form, the behavior is undefined.

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>** (7.1.6).

6.1.5 Operators

Syntax

operator: one of

```
[ ] ( ) . ->
++ -- & * + - ~ ! sizeof
/ % << >> < > <= >= == != ^ | && ||
? :
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
```

Constraints

The operators **[]**, **()**, and **? :** shall occur in pairs, possibly separated by expressions. The operators **#** and **##** shall occur in macro-defining preprocessing directives only.

Semantics

An operator specifies an operation to be performed (an *evaluation*) that yields a value, or yields a designator, or produces a side effect, or a combination thereof. An *operand* is an entity on which an operator acts.

Forward references: expressions (6.3), macro replacement (6.8.3).

²⁴ 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.

6.1.6 Punctuators

Syntax

punctuator: one of
 [] () { } * , : = ; ... #

Constraints

The punctuators [], (), and { } shall occur (after translation phase 4) in pairs, possibly separated by expressions, declarations, or statements. The punctuator # shall occur in preprocessing directives only.

Semantics

A punctuator is a symbol that has independent syntactic and semantic significance but does not specify an operation to be performed that yields a value. Depending on context, the same symbol may also represent an operator or part of an operator.

Forward references: expressions (6.3), declarations (6.5), preprocessing directives (6.8), statements (6.6).

6.1.7 Header names

Syntax

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 "

Constraints

Header name preprocessing tokens shall only appear within a **#include** preprocessing directive.

Semantics

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 6.8.2.

If the characters ', \, ", or /* occur in the sequence between the < and > delimiters, the behavior is undefined. Similarly, if the characters ', \, or /* occur in the sequence between the " delimiters, the behavior is undefined.²⁵

²⁵ Thus, sequences of characters that resemble escape sequences cause undefined behavior.

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 (6.8.2).

6.1.8 Preprocessing numbers**Syntax**

```
pp-number:
    digit
    . digit
    pp-number digit
    pp-number nondigit
    pp-number e sign
    pp-number E sign
    pp-number .
```

Description

A preprocessing number begins with a digit optionally preceded by a period (.) and may be followed by letters, underscores, digits, periods, and **e+**, **e-**, **E+**, or **E-** character sequences.

Preprocessing number tokens lexically include all floating and integer constant tokens.

Semantics

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.

6.1.9 Comments

Except within a character constant, a string literal, or a comment, the characters **/*** introduce a comment. The contents of a comment are examined only to identify multibyte characters and to find the characters ***/** that terminate it.²⁶

²⁶ Thus, comments do not nest.

6.2 Conversions

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 6.2.1.5 summarizes the conversions performed by most ordinary operators; it is supplemented as required by the discussion of each operator in 6.3.

Conversion of an operand value to a compatible type causes no change to the value or the representation.

Forward references: cast operators (6.3.4).

6.2.1 Arithmetic operands

6.2.1.1 Characters and integers

A **char**, a **short int**, or an **int** bit-field, or their signed or unsigned varieties, or an enumeration type, may be used in an expression wherever an **int** or **unsigned int** may be used. 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 *integral promotions*.²⁷ All other arithmetic types are unchanged by the integral promotions.

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

Forward references: enumeration specifiers (6.5.2.2), structure and union specifiers (6.5.2.1).

6.2.1.2 Signed and unsigned integers

When a value with integral type is converted to another integral type, if the value can be represented by the new type, its value is unchanged.

When a signed integer is converted to an unsigned integer with equal or greater size, if the value of the signed integer is nonnegative, its value is unchanged. Otherwise: if the unsigned integer has greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned integer; the value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.²⁸

When a value with integral type is demoted to an unsigned integer with smaller size, the result is the nonnegative remainder on division by the number one greater than the largest unsigned number that can be represented in the type with smaller size. When a value with integral type is demoted to a signed integer with smaller size, or an unsigned integer is converted to its corresponding signed integer, if the value cannot be represented the result is implementation-defined.

²⁷ The integral 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.

²⁸ In a two's-complement representation, there is no actual change in the bit pattern except filling the high-order bits with copies of the sign bit if the unsigned integer has greater size.

6.2.1.3 Floating and integral

When a value of floating type is converted to integral type, the fractional part is discarded. If the value of the integral part cannot be represented by the integral type, the behavior is undefined.²⁹

When a value of integral type is converted to floating type, 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 value, chosen in an implementation-defined manner.

6.2.1.4 Floating types

When a **float** is promoted to **double** or **long double**, or a **double** is promoted to **long double**, its value is unchanged.

When a **double** is demoted to **float** or a **long double** to **double** or **float**, if the value being converted is outside the range of values that can be represented, the behavior is undefined. 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 value, chosen in an implementation-defined manner.

6.2.1.5 Usual arithmetic conversions

Many binary operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*:

First, if either operand has type **long double**, the other operand is converted to **long double**.

Otherwise, if either operand has type **double**, the other operand is converted to **double**.

Otherwise, if either operand has type **float**, the other operand is converted to **float**.

Otherwise, the integral promotions are performed on both operands. Then the following rules are applied:

If either operand has type **unsigned long int**, the other operand is converted to **unsigned long int**.

Otherwise, if one operand has type **long int** and the other has type **unsigned int**, if a **long int** can represent all values of an **unsigned int**, the operand of type **unsigned int** is converted to **long int**; if a **long int** cannot represent all the values of an **unsigned int**, both operands are converted to **unsigned long int**.

Otherwise, if either operand has type **long int**, the other operand is converted to **long int**.

Otherwise, if either operand has type **unsigned int**, the other operand is converted to **unsigned int**.

Otherwise, both operands have type **int**.

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.³⁰

²⁹ The remainder operation performed when a value of integral type is converted to unsigned type need not be performed when a value of floating type is converted to unsigned type. Thus, the range of portable floating values is $(-1, \text{Utype_MAX}+1)$.

³⁰ The cast and assignment operators still must perform their specified conversions, as described in 6.2.1.3 and 6.2.1.4.

6.2.2 Other operands

6.2.2.1 Lvalues and function designators

An *lvalue* is an expression (with an object type or an incomplete type other than **void**) that designates an object.³¹ 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 of all contained structures or unions) with a const-qualified type.

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.

Except when it is the operand of the **sizeof** operator or the unary **&** operator, or is a character string literal used to initialize an array of character type, or is a wide string literal used to initialize an array with element type compatible with **wchar_t**, an lvalue that has type “array of *type*” is converted to an expression that has type “pointer to *type*” that points to the initial element of the array object and is not an lvalue.

A *function designator* is an expression that has function type. Except when it is the operand of the **sizeof** operator³² 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 (6.3.3.2), assignment operators (6.3.16), common definitions **<stddef.h>** (7.1.6), initialization (6.5.7), postfix increment and decrement operators (6.3.2.4), prefix increment and decrement operators (6.3.3.1), the **sizeof** operator (6.3.3.4), structure and union members (6.3.2.3).

6.2.2.2 void

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 occurs in a context where a void expression is required, its value or designator is discarded. (A void expression is evaluated for its side effects.)

6.2.2.3 Pointers

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.

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.

³¹ The name “lvalue” comes originally from the assignment expression **E1 = E2**, in which the left operand **E1** must 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.

³² Because this conversion does not occur, the operand of the **sizeof** operator remains a function designator and violates the constraint in 6.3.3.4.

An integral constant expression with the value 0, or such an expression cast to type **void ***, is called a *null pointer constant*.³³ If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. Such a pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

Two null pointers, converted through possibly different sequences of casts to pointer types, shall compare equal.

Forward references: cast operators (6.3.4), equality operators (6.3.9), simple assignment (6.3.16.1).

³³ The macro **NULL** is defined in **<stddef.h>** as a null pointer constant; see 7.1.6.

6.3 Expressions

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.

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 accessed only to determine the value to be stored.³⁴

Except as indicated by the syntax³⁵ or otherwise specified later (for the function-call operator `()`, `&&`, `||`, `?:`, and comma operators), the order of evaluation of subexpressions and the order in which side effects take place are both unspecified.

Some operators (the unary operator `~`, and the binary operators `<<`, `>>`, `&`, `^`, and `|`, collectively described as *bitwise operators*) shall have operands that have integral type. These operators return values that depend on the internal representations of integers, and thus have implementation-defined aspects for signed types.

If an *exception* 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.

An object shall have its stored value accessed only by an lvalue that has one of the following types:³⁶

- the declared type of the object,
- a qualified version of the declared type of the object,
- a type that is the signed or unsigned type corresponding to the declared type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the declared 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.

³⁴ This paragraph renders undefined statement expressions such as

```
i = ++i + 1;
```

while allowing

```
i = i + 1;
```

³⁵ 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 (6.3.6) shall be those expressions defined in 6.3.1 through 6.3.6. The exceptions are cast expressions (6.3.4) as operands of unary operators (6.3.3), and an operand contained between any of the following pairs of operators: grouping parentheses `()` (6.3.1), subscripting brackets `[]` (6.3.2.1), function-call parentheses `()` (6.3.2.2), and the conditional operator `?:` (6.3.15).

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.

³⁶ The intent of this list is to specify those circumstances in which an object may or may not be aliased.

6.3.1 Primary expressions

Syntax

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

Semantics

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).

A constant is a primary expression. Its type depends on its form and value, as detailed in 6.1.3.

A string literal is a primary expression. It is an lvalue with type as detailed in 6.1.4.

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 (6.5).

6.3.2 Postfix operators

Syntax

postfix-expression:
primary-expression
postfix-expression [expression]
postfix-expression (argument-expression-list_{opt})
postfix-expression . identifier
postfix-expression -> identifier
postfix-expression ++
postfix-expression --

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

6.3.2.1 Array subscripting

Constraints

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

Semantics

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).

Successive subscript operators designate an element of a multidimensional array object. If **E** is an *n*-dimensional array (*n* ≥ 2) with dimensions *i* × *j* × ... × *k*, then **E** (used as other than an lvalue) is converted to a pointer to an (*n* − 1)-dimensional array with dimensions *j* × ... × *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).

Example

Consider the array object defined by the declaration

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

Here **x** is a 3×5 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 in turn is converted to a pointer to the first of the **ints**, so **x[i][j]** yields an **int**.

Forward references: additive operators (6.3.6), address and indirection operators (6.3.3.2), array declarators (6.5.4.2).

6.3.2.2 Function calls

Constraints

The expression that denotes the called function³⁷ shall have type pointer to function returning **void** or returning an object type other than an array type.

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

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.

If the expression that precedes the parenthesized argument list in a function call consists solely of an identifier, and if no declaration is visible for this identifier, the identifier is implicitly declared exactly as if, in the innermost block containing the function call, the declaration

```
extern int identifier();
```

appeared.³⁸

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.³⁹ The value of the function call expression is specified in 6.6.6.4.

³⁷ Most often, this is the result of converting an identifier that is a function designator.

³⁸ That is, an identifier with block scope declared to have external linkage with type function without parameter information and returning an **int**. If in fact it is not defined as having type “function returning **int**,” the behavior is undefined.

³⁹ 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 converted to a parameter with a pointer type as described in 6.7.1.

If the expression that denotes the called function has a type that does not include a prototype, the integral 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 agree with the number of 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. If the function is defined with a type that includes a prototype, and the types of the arguments after promotion are not compatible with the types of the parameters, or if the prototype ends with an ellipsis (`, ...`), the behavior is undefined.

If the expression that denotes the called function has a type that includes a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters. 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. 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.

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.

The order of evaluation of the function designator, the arguments, and subexpressions within the arguments is unspecified, but there is a sequence point before the actual call.

Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

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 shall be completed before the function pointed to by **pf[f1()]** is entered.

Forward references: function declarators (including prototypes) (6.5.4.3), function definitions (6.7.1), the **return** statement (6.6.6.4), simple assignment (6.3.16.1).

6.3.2.3 Structure and union members

Constraints

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.

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

A postfix expression followed by a dot `.` 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.

A postfix expression followed by an arrow $\rightarrow$ 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.⁴⁰ 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.

With one exception, if a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined.⁴¹ 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. 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.

Examples

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.
2. 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)
    /*...*/ sin(u.nf.doublenode) /*...*/
```

Forward references: address and indirection operators (6.3.3.2), structure and union specifiers (6.5.2.1).

6.3.2.4 Postfix increment and decrement operators

Constraints

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

⁴⁰ If $\&E$ is a valid pointer expression (where $\&$ is the “address-of” operator, which generates a pointer to its operand), the expression $(\&E) \rightarrow MOS$ is the same as $E.MOS$.

⁴¹ The “byte orders” for scalar types are invisible to isolated programs that do not indulge in type punning (for example, by assigning to one member of a union and inspecting the storage by accessing another member that is an appropriately sized array of character type), but must be accounted for when conforming to externally imposed storage layouts.