BS ISO/IEC 14882:2017

BSI Standards Publication

Programming languages — C++

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

National foreword

This British Standard is the UK implementation of ISO/IEC 14882:2017. It supersedes BS ISO/IEC 14882:2014, which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environments and system software interfaces.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2018
Published by BSI Standards Limited 2018

ISBN 978 0 580 90582 7

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 January 2018.

Amendments/corrigenda issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

INTERNATIONAL STANDARD

ISO/IEC 14882

Fifth edition
2017-12

Programming languages — C++

Langages de programmation — C++

Reference number
ISO/IEC 14882:2017(E)

Documents
Document Sharing Hub

© ISO/IEC 2017

Full document details: BS ISO/IEC 14882:2017

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2017, Published in Switzerland

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

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

© ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

Contents

Foreword **xi**

1 Scope **1**

2 Normative references **2**

3 Terms and definitions **3**

4 General principles **7**
4.1 Implementation compliance . . . 7
4.2 Structure of this document . . . 8
4.3 Syntax notation . . . 8
4.4 The C++ memory model . . . 8
4.5 The C++ object model . . . 9
4.6 Program execution . . . 11
4.7 Multi-threaded executions and data races . . . 15
4.8 Acknowledgments . . . 20

5 Lexical conventions **22**
5.1 Separate translation . . . 22
5.2 Phases of translation . . . 22
5.3 Character sets . . . 23
5.4 Preprocessing tokens . . . 24
5.5 Alternative tokens . . . 25
5.6 Tokens . . . 25
5.7 Comments . . . 26
5.8 Header names . . . 26
5.9 Preprocessing numbers . . . 26
5.10 Identifiers . . . 27
5.11 Keywords . . . 28
5.12 Operators and punctuators . . . 29
5.13 Literals . . . 29

6 Basic concepts **39**
6.1 Declarations and definitions . . . 39
6.2 One-definition rule . . . 41
6.3 Scope . . . 44
6.4 Name lookup . . . 50
6.5 Program and linkage . . . 63
6.6 Start and termination . . . 66
6.7 Storage duration . . . 70
6.8 Object lifetime . . . 74
6.9 Types . . . 77
6.10 Lvalues and rvalues . . . 83
6.11 Alignment . . . 84

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

7 Standard conversions **86**
7.1 Lvalue-to-rvalue conversion . . . 87
7.2 Array-to-pointer conversion . . . 87
7.3 Function-to-pointer conversion . . . 88
7.4 Temporary materialization conversion . . . 88
7.5 Qualification conversions . . . 88
7.6 Integral promotions . . . 89
7.7 Floating-point promotion . . . 89
7.8 Integral conversions . . . 89
7.9 Floating-point conversions . . . 90
7.10 Floating-integral conversions . . . 90
7.11 Pointer conversions . . . 90
7.12 Pointer to member conversions . . . 90
7.13 Function pointer conversions . . . 91
7.14 Boolean conversions . . . 91
7.15 Integer conversion rank . . . 91

8 Expressions **93**
8.1 Primary expressions . . . 96
8.2 Postfix expressions . . . 109
8.3 Unary expressions . . . 120
8.4 Explicit type conversion (cast notation) . . . 129
8.5 Pointer-to-member operators . . . 130
8.6 Multiplicative operators . . . 131
8.7 Additive operators . . . 131
8.8 Shift operators . . . 132
8.9 Relational operators . . . 133
8.10 Equality operators . . . 133
8.11 Bitwise AND operator . . . 135
8.12 Bitwise exclusive OR operator . . . 135
8.13 Bitwise inclusive OR operator . . . 135
8.14 Logical AND operator . . . 135
8.15 Logical OR operator . . . 135
8.16 Conditional operator . . . 136
8.17 Throwing an exception . . . 137
8.18 Assignment and compound assignment operators . . . 137
8.19 Comma operator . . . 138
8.20 Constant expressions . . . 139

9 Statements **144**
9.1 Labeled statement . . . 145
9.2 Expression statement . . . 145
9.3 Compound statement or block . . . 145
9.4 Selection statements . . . 145
9.5 Iteration statements . . . 148
9.6 Jump statements . . . 150
9.7 Declaration statement . . . 152
9.8 Ambiguity resolution . . . 153

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

10 Declarations **155**
10.1 Specifiers . . . 157
10.2 Enumeration declarations . . . 174
10.3 Namespaces . . . 178
10.4 The `asm` declaration . . . 191
10.5 Linkage specifications . . . 191
10.6 Attributes . . . 194

11 Declarators **201**
11.1 Type names . . . 202
11.2 Ambiguity resolution . . . 203
11.3 Meaning of declarators . . . 204
11.4 Function definitions . . . 216
11.5 Structured binding declarations . . . 219
11.6 Initializers . . . 220

12 Classes **237**
12.1 Class names . . . 239
12.2 Class members . . . 241
12.3 Unions . . . 251
12.4 Local class declarations . . . 254

13 Derived classes **255**
13.1 Multiple base classes . . . 256
13.2 Member name lookup . . . 258
13.3 Virtual functions . . . 261
13.4 Abstract classes . . . 265

14 Member access control **267**
14.1 Access specifiers . . . 268
14.2 Accessibility of base classes and base class members . . . 269
14.3 Friends . . . 272
14.4 Protected member access . . . 275
14.5 Access to virtual functions . . . 276
14.6 Multiple access . . . 276
14.7 Nested classes . . . 276

15 Special member functions **278**
15.1 Constructors . . . 278
15.2 Temporary objects . . . 281
15.3 Conversions . . . 283
15.4 Destructors . . . 286
15.5 Free store . . . 289
15.6 Initialization . . . 291
15.7 Construction and destruction . . . 298
15.8 Copying and moving class objects . . . 301

16 Overloading **309**
16.1 Overloadable declarations . . . 309
16.2 Declaration matching . . . 311
16.3 Overload resolution . . . 312

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

16.4 Address of overloaded function . . . 333
16.5 Overloaded operators . . . 334
16.6 Built-in operators . . . 339

17 Templates **342**
17.1 Template parameters . . . 343
17.2 Names of template specializations . . . 347
17.3 Template arguments . . . 348
17.4 Type equivalence . . . 354
17.5 Template declarations . . . 355
17.6 Name resolution . . . 373
17.7 Template instantiation and specialization . . . 388
17.8 Function template specializations . . . 400
17.9 Deduction guides . . . 421

18 Exception handling **422**
18.1 Throwing an exception . . . 423
18.2 Constructors and destructors . . . 425
18.3 Handling an exception . . . 425
18.4 Exception specifications . . . 427
18.5 Special functions . . . 430

19 Preprocessing directives **432**
19.1 Conditional inclusion . . . 433
19.2 Source file inclusion . . . 435
19.3 Macro replacement . . . 436
19.4 Line control . . . 441
19.5 Error directive . . . 442
19.6 Pragma directive . . . 442
19.7 Null directive . . . 442
19.8 Predefined macro names . . . 442
19.9 Pragma operator . . . 444

20 Library introduction **445**
20.1 General . . . 445
20.2 The C standard library . . . 446
20.3 Definitions . . . 446
20.4 Method of description (Informative) . . . 449
20.5 Library-wide requirements . . . 454

21 Language support library **476**
21.1 General . . . 476
21.2 Common definitions . . . 476
21.3 Implementation properties . . . 481
21.4 Integer types . . . 490
21.5 Start and termination . . . 491
21.6 Dynamic memory management . . . 492
21.7 Type identification . . . 500
21.8 Exception handling . . . 502
21.9 Initializer lists . . . 507
21.10 Other runtime support . . . 508

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

22 Diagnostics library **511**
22.1 General . . . 511
22.2 Exception classes . . . 511
22.3 Assertions . . . 515
22.4 Error numbers . . . 515
22.5 System error support . . . 517

23 General utilities library **528**
23.1 General . . . 528
23.2 Utility components . . . 528
23.3 Compile-time integer sequences . . . 536
23.4 Pairs . . . 537
23.5 Tuples . . . 541
23.6 Optional objects . . . 553
23.7 Variants . . . 567
23.8 Storage for any type . . . 580
23.9 Bitsets . . . 586
23.10 Memory . . . 592
23.11 Smart pointers . . . 607
23.12 Memory resources . . . 634
23.13 Class template `scoped_allocator_adaptor` . . . 645
23.14 Function objects . . . 651
23.15 Metaprogramming and type traits . . . 675
23.16 Compile-time rational arithmetic . . . 699
23.17 Time utilities . . . 702
23.18 Class `type_index` . . . 719
23.19 Execution policies . . . 720

24 Strings library **723**
24.1 General . . . 723
24.2 Character traits . . . 723
24.3 String classes . . . 729
24.4 String view classes . . . 762
24.5 Null-terminated sequence utilities . . . 772

25 Localization library **778**
25.1 General . . . 778
25.2 Header `<locale>` synopsis . . . 778
25.3 Locales . . . 780
25.4 Standard `locale` categories . . . 787
25.5 C library locales . . . 825

26 Containers library **826**
26.1 General . . . 826
26.2 Container requirements . . . 826
26.3 Sequence containers . . . 864
26.4 Associative containers . . . 896
26.5 Unordered associative containers . . . 918
26.6 Container adaptors . . . 942

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

27 Iterators library **952**
27.1 General . . . 952
27.2 Iterator requirements . . . 952
27.3 Header `<iterator>` synopsis . . . 958
27.4 Iterator primitives . . . 961
27.5 Iterator adaptors . . . 964
27.6 Stream iterators . . . 977
27.7 Range access . . . 984
27.8 Container access . . . 985

28 Algorithms library **986**
28.1 General . . . 986
28.2 Header `<algorithm>` synopsis . . . 986
28.3 Algorithms requirements . . . 1005
28.4 Parallel algorithms . . . 1006
28.5 Non-modifying sequence operations . . . 1009
28.6 Mutating sequence operations . . . 1017
28.7 Sorting and related operations . . . 1027
28.8 C library algorithms . . . 1046

29 Numerics library **1047**
29.1 General . . . 1047
29.2 Definitions . . . 1047
29.3 Numeric type requirements . . . 1047
29.4 The floating-point environment . . . 1048
29.5 Complex numbers . . . 1049
29.6 Random number generation . . . 1059
29.7 Numeric arrays . . . 1102
29.8 Generalized numeric operations . . . 1122
29.9 Mathematical functions for floating-point types . . . 1136

30 Input/output library **1153**
30.1 General . . . 1153
30.2 Iostreams requirements . . . 1154
30.3 Forward declarations . . . 1154
30.4 Standard iostream objects . . . 1156
30.5 Iostreams base classes . . . 1158
30.6 Stream buffers . . . 1175
30.7 Formatting and manipulators . . . 1184
30.8 String-based streams . . . 1211
30.9 File-based streams . . . 1221
30.10 File systems . . . 1235
30.11 C library files . . . 1288

31 Regular expressions library **1292**
31.1 General . . . 1292
31.2 Definitions . . . 1292
31.3 Requirements . . . 1293
31.4 Header `<regex>` synopsis . . . 1295
31.5 Namespace `std::regex_constants` . . . 1301
31.6 Class `regex_error` . . . 1304

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

31.7 Class template `regex_traits` . . . 1305
31.8 Class template `basic_regex` . . . 1307
31.9 Class template `sub_match` . . . 1313
31.10 Class template `match_results` . . . 1318
31.11 Regular expression algorithms . . . 1324
31.12 Regular expression iterators . . . 1329
31.13 Modified ECMAScript regular expression grammar . . . 1335

32 Atomic operations library **1338**
32.1 General . . . 1338
32.2 Header `<atomic>` synopsis . . . 1338
32.3 Type aliases . . . 1342
32.4 Order and consistency . . . 1342
32.5 Lock-free property . . . 1344
32.6 Class template `atomic` . . . 1345
32.7 Non-member functions . . . 1352
32.8 Flag type and operations . . . 1352
32.9 Fences . . . 1353

33 Thread support library **1355**
33.1 General . . . 1355
33.2 Requirements . . . 1355
33.3 Threads . . . 1358
33.4 Mutual exclusion . . . 1363
33.5 Condition variables . . . 1384
33.6 Futures . . . 1391

A Grammar summary **1408**
A.1 Keywords . . . 1408
A.2 Lexical conventions . . . 1408
A.3 Basic concepts . . . 1413
A.4 Expressions . . . 1413
A.5 Statements . . . 1417
A.6 Declarations . . . 1418
A.7 Declarators . . . 1422
A.8 Classes . . . 1424
A.9 Derived classes . . . 1425
A.10 Special member functions . . . 1426
A.11 Overloading . . . 1426
A.12 Templates . . . 1426
A.13 Exception handling . . . 1427
A.14 Preprocessing directives . . . 1428

B Implementation quantities **1430**

C Compatibility **1432**
C.1 C++ and ISO C . . . 1432
C.2 C++ and ISO C++ 2003 . . . 1441
C.3 C++ and ISO C++ 2011 . . . 1447
C.4 C++ and ISO C++ 2014 . . . 1449
C.5 C standard library . . . 1453

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

D Compatibility features **1456**
D.1 Redeclaration of `static constexpr` data members . . . 1456
D.2 Implicit declaration of copy functions . . . 1456
D.3 Deprecated exception specifications . . . 1456
D.4 C++ standard library headers . . . 1456
D.5 C standard library headers . . . 1457
D.6 `char*` streams . . . 1457
D.7 `uncaught_exception` . . . 1466
D.8 Old adaptable function bindings . . . 1466
D.9 The default allocator . . . 1471
D.10 Raw storage iterator . . . 1472
D.11 Temporary buffers . . . 1473
D.12 Deprecated type traits . . . 1474
D.13 Deprecated iterator primitives . . . 1475
D.14 Deprecated `shared_ptr` observers . . . 1475
D.15 Deprecated standard code conversion facets . . . 1475
D.16 Deprecated convenience conversion interfaces . . . 1477

Bibliography **1482**

Cross references **1483**

Cross references from ISO C++ 2014 **1504**

Index **1506**

Index of grammar productions **1539**

Index of library names **1543**

Index of implementation-defined behavior **1601**

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

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

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

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

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

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

This fifth edition cancels and replaces the fourth edition (ISO/IEC 14882:2014), which has been technically revised.

The main changes compared to the previous edition are as follows:

— expression evaluation order is specified in more cases

— removal of trigraphs

— adjustments to value categories resulting in copy elision being mandatory

— additional character and floating point literal syntaxes

— lambda expressions extended to permit capture of `*this` and use in constant expressions

— initializer statements for `if` and `switch` statements

— addition of constexpr if statements

— range-based `for` statement generalized to support heterogeneous `begin` and `end` types

— addition of structured bindings

— addition of inline variables

— list initialization extended to support enumerations and aggregates with base classes

— message in `static_assert` is now optional

— addition of nested namespace definition syntax

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

- extended support for attributes
- exception specifications are now part of function types
- template argument deduction is now supported for class templates
- addition of fold expressions
- pack expansion can be performed on using declarations
- permitted forms of template parameters and template arguments have been generalized
- dynamic allocation is supported for over-aligned types
- preprocessor can detect presence of header files with `__has_include`
- new utility functions, types, and templates in the standard library, including
  - an `any` type
  - an `optional` class template
  - a `variant` class template
  - a `clamp` function
  - a `std::byte` type
  - a `not_fn` function
  - a `void_t` alias template
  - `conjunction`, `disjunction`, and `negation` templates
  - an `invoke` function, and `is_invocable` and `invoke_result` type traits
  - an `is_swappable` type trait
- extended constant expression evaluation support in the standard library
- elementary conversion functions between strings and numeric types added
- constructors for `pair` and `tuple` are conditionally-explicit
- `shared_ptr`s of array types now supported
- additional algorithms for managing uninitialized memory
- addition of polymorphic memory resources
- addition of substring search facilities providing the Boyer-Moore and Boyer-Moore-Horspool search algorithms
- addition of variable templates for type traits
- addition of a non-owning string view template
- ability to splice elements between containers for maps and sets
- better support for element insertion in unique-key maps
- support for incomplete types in containers
- addition of parallel algorithms
- addition of `sample` algorithm
- addition of mathematical special functions, and `gcd`, `lcm`, and three-argument `hypot` functions
- addition of support for operations on file systems
- addition of shared mutexes and variadic lock guards
- removal of deprecated features

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

1 Scope [intro.scope]

1 This document specifies requirements for implementations of the C++ programming language. The first such requirement is that they implement the language, so this document also defines C++. Other requirements and relaxations of the first requirement appear at various places within this document.

2 C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899:2011 *Programming languages — C* (hereinafter referred to as the *C standard*). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

2 Normative references [intro.refs]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — Ecma International, *ECMAScript Language Specification*, Standard Ecma-262, third edition, 1999.

(1.2) — ISO/IEC 2382 (all parts), *Information technology — Vocabulary*

(1.3) — ISO/IEC 9899:2011, *Programming languages — C*

(1.4) — ISO/IEC 9945:2003, *Information Technology — Portable Operating System Interface (POSIX)*

(1.5) — ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*

(1.6) — ISO/IEC/IEEE 60559:2011, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

(1.7) — ISO 80000-2:2009, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*

2 The library described in Clause 7 of ISO/IEC 9899:2011 is hereinafter called the *C standard library*.[1]

3 The operating system interface described in ISO/IEC 9945:2003 is hereinafter called *POSIX*.

4 The ECMAScript Language Specification described in Standard Ecma-262 is hereinafter called *ECMA-262*.

1) With the qualifications noted in Clauses 21 through 33 and in C.5, the C standard library is a subset of the C++ standard library.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

3 Terms and definitions [intro.defs]

1 For the purposes of this document, the terms and definitions given in ISO/IEC 2382-1:1993, the terms, definitions, and symbols given in ISO 80000-2:2009, and the following apply.

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

(2.1) — ISO Online browsing platform: available at http://www.iso.org/obp

(2.2) — IEC Electropedia: available at http://www.electropedia.org/

3 20.3 defines additional terms that are used only in Clauses 20 through 33 and Annex D.

4 Terms that are used only in a small portion of this document are defined where they are used and italicized where they are defined.

3.1 **[defns.access]**
access
⟨execution-time action⟩ read or modify the value of an object

3.2 **[defns.argument]**
argument
⟨function call expression⟩ expression in the comma-separated list bounded by the parentheses (8.2.2)

3.3 **[defns.argument.macro]**
argument
⟨function-like macro⟩ sequence of preprocessing tokens in the comma-separated list bounded by the parentheses (19.3)

3.4 **[defns.argument.throw]**
argument
⟨throw expression⟩ operand of `throw` (8.17)

3.5 **[defns.argument.templ]**
argument
⟨template instantiation⟩ *constant-expression*, *type-id*, or *id-expression* in the comma-separated list bounded by the angle brackets (17.3)

3.6 **[defns.block]**
block
wait for some condition (other than for the implementation to execute the execution steps of the thread of execution) to be satisfied before continuing execution past the blocking operation

3.7 **[defns.cond.supp]**
conditionally-supported
program construct that an implementation is not required to support

[*Note 1 to entry:* Each implementation documents all conditionally-supported constructs that it does not support. — *end note*]

3.8 **[defns.diagnostic]**
diagnostic message
message belonging to an implementation-defined subset of the implementation's output messages

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

3.9 **[defns.dynamic.type]**

dynamic type

⟨glvalue⟩ type of the most derived object (4.5) to which the glvalue refers

[*Example:* If a pointer (11.3.1) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (Clause 13), the dynamic type of the expression `*p` is "`D`". References (11.3.2) are treated similarly. — *end example*]

3.10 **[defns.dynamic.type.prvalue]**

dynamic type

⟨prvalue⟩ static type of the prvalue expression

3.11 **[defns.ill.formed]**

ill-formed program

program that is not well-formed (3.29)

3.12 **[defns.impl.defined]**

implementation-defined behavior

behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents

3.13 **[defns.impl.limits]**

implementation limits

restrictions imposed upon programs by the implementation

3.14 **[defns.locale.specific]**

locale-specific behavior

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

3.15 **[defns.multibyte]**

multibyte character

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

[*Note 1 to entry:* The extended character set is a superset of the basic character set (5.3). — *end note*]

3.16 **[defns.parameter]**

parameter

⟨function or catch clause⟩ object or reference declared as part of a function declaration or definition or in the catch clause of an exception handler that acquires a value on entry to the function or handler

3.17 **[defns.parameter.macro]**

parameter

⟨function-like macro⟩ identifier from the comma-separated list bounded by the parentheses immediately following the macro name

3.18 **[defns.parameter.templ]**

parameter

⟨template⟩ member of a *template-parameter-list*

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

3.19 **[defns.signature]**

signature

⟨function⟩ name, parameter type list (11.3.5), and enclosing namespace (if any)

[*Note 1 to entry:* Signatures are used as a basis for name mangling and linking. — *end note*]

3.20 **[defns.signature.templ]**

signature

⟨function template⟩ name, parameter type list (11.3.5), enclosing namespace (if any), return type, and template parameter list

3.21 **[defns.signature.spec]**

signature

⟨function template specialization⟩ signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced)

3.22 **[defns.signature.member]**

signature

⟨class member function⟩ name, parameter type list (11.3.5), class of which the function is a member, *cv*-qualifiers (if any), and *ref-qualifier* (if any)

3.23 **[defns.signature.member.templ]**

signature

⟨class member function template⟩ name, parameter type list (11.3.5), class of which the function is a member, *cv*-qualifiers (if any), *ref-qualifier* (if any), return type (if any), and template parameter list

3.24 **[defns.signature.member.spec]**

signature

⟨class member function template specialization⟩ signature of the member function template of which it is a specialization and its template arguments (whether explicitly specified or deduced)

3.25 **[defns.static.type]**

static type

type of an expression (6.9) resulting from analysis of the program without considering execution semantics

[*Note 1 to entry:* The static type of an expression depends only on the form of the program in which the expression appears, and does not change while the program is executing. — *end note*]

3.26 **[defns.unblock]**

unblock

satisfy a condition that one or more blocked threads of execution are waiting for

3.27 **[defns.undefined]**

undefined behavior

behavior for which this document imposes no requirements

[*Note 1 to entry:* Undefined behavior may be expected when this document omits any explicit definition of behavior or when a program uses an erroneous construct or erroneous data. 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). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. Evaluation of a constant expression never exhibits behavior explicitly specified as undefined (8.20). — *end note*]

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

3.28 **[defns.unspecified]**

unspecified behavior

behavior, for a well-formed program construct and correct data, that depends on the implementation

[*Note 1 to entry:* The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this document. — *end note*]

3.29 **[defns.well.formed]**

well-formed program

C++ program constructed according to the syntax rules, diagnosable semantic rules, and the one-definition rule (6.2).

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

4 General principles [intro]

4.1 Implementation compliance [intro.compliance]

1 The set of *diagnosable rules* consists of all syntactic and semantic rules in this document except for those rules containing an explicit notation that "no diagnostic is required" or which are described as resulting in "undefined behavior".

2 Although this document states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:

(2.1) — If a program contains no violations of the rules in this document, a conforming implementation shall, within its resource limits, accept and correctly execute[2] that program.

(2.2) — If a program contains a violation of any diagnosable rule or an occurrence of a construct described in this document as "conditionally-supported" when the implementation does not support that construct, a conforming implementation shall issue at least one diagnostic message.

(2.3) — If a program contains a violation of a rule for which no diagnostic is required, this document places no requirement on implementations with respect to that program.

[*Note:* During template argument deduction and substitution, certain constructs that in other contexts require a diagnostic are treated differently; see 17.8.2. — *end note*]

3 For classes and class templates, the library Clauses specify partial definitions. Private members (Clause 14) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library Clauses.

4 For functions, function templates, objects, and values, the library Clauses specify declarations. Implementations shall supply definitions consistent with the descriptions in the library Clauses.

5 The names defined in the library have namespace scope (10.3). A C++ translation unit (5.2) obtains access to these names by including the appropriate standard library header (19.2).

6 The templates, classes, functions, and objects in the library have external linkage (6.5). The implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (5.2).

7 Two kinds of implementations are defined: a *hosted implementation* and a *freestanding implementation*. For a hosted implementation, this document defines the set of available libraries. A freestanding implementation is one in which execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that includes certain language-support libraries (20.5.1.3).

8 A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any well-formed program. Implementations are required to diagnose programs that use such extensions that are ill-formed according to this document. Having done so, however, they can compile and execute such programs.

9 Each implementation shall include documentation that identifies all conditionally-supported constructs that it does not support and defines all locale-specific characteristics.[3]

2) "Correct execution" can include undefined behavior, depending on the data being processed; see Clause 3 and 4.6.
3) This documentation also defines implementation-defined behavior; see 4.6.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

4.2 Structure of this document [intro.structure]

1 Clauses 5 through 19 describe the C++ programming language. That description includes detailed syntactic specifications in a form described in 4.3. For convenience, Annex A repeats all such syntactic specifications.

2 Clauses 21 through 33 and Annex D (the *library clauses*) describe the C++ standard library. That description includes detailed descriptions of the entities and macros that constitute the library, in a form described in Clause 20.

3 Annex B recommends lower bounds on the capacity of conforming implementations.

4 Annex C summarizes the evolution of C++ since its first published description, and explains in detail the differences between C++ and C. Certain features of C++ exist solely for compatibility purposes; Annex D describes those features.

5 Throughout this document, each example is introduced by "[*Example:* " and terminated by " — *end example*]". Each note is introduced by "[*Note:* " and terminated by " — *end note*]". Examples and notes may be nested.

4.3 Syntax notation [syntax]

1 In the syntax notation used in this document, syntactic categories are indicated by *italic* type, and literal words and characters in `constant width` type. Alternatives are listed on separate lines except in a few cases where a long set of alternatives is marked by the phrase "one of". If the text of an alternative is too long to fit on a line, the text is continued on subsequent lines indented from the first one. An optional terminal or non-terminal symbol is indicated by the subscript "$_{opt}$", so

`{` *expression*$_{opt}$ `}`

indicates an optional expression enclosed in braces.

2 Names for syntactic categories have generally been chosen according to the following rules:

(2.1) — *X-name* is a use of an identifier in a context that determines its meaning (e.g., *class-name*, *typedef-name*).

(2.2) — *X-id* is an identifier with no context-dependent meaning (e.g., *qualified-id*).

(2.3) — *X-seq* is one or more *X*'s without intervening delimiters (e.g., *declaration-seq* is a sequence of declarations).

(2.4) — *X-list* is one or more *X*'s separated by intervening commas (e.g., *identifier-list* is a sequence of identifiers separated by commas).

4.4 The C++ memory model [intro.memory]

1 The fundamental storage unit in the C++ memory model is the *byte*. A byte is at least large enough to contain any member of the basic execution character set (5.3) and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits,[4] 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*. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

2 [*Note:* The representation of types is described in 6.9. — *end note*]

3 A *memory location* is either an object of scalar type or a maximal sequence of adjacent bit-fields all having nonzero width. [*Note:* Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. — *end note*] Two or more threads of execution (4.7) can access separate memory locations without interfering with each other.

4 [*Note:* Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields,

4) The number of bits in a byte is reported by the macro `CHAR_BIT` in the header `<climits>`.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of nonzero width. — *end note*]

5 [*Example:* A structure declared as

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

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

4.5 The C++ object model [intro.object]

1 The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is created by a definition (6.1), by a *new-expression* (8.3.4), when implicitly changing the active member of a union (12.3), or when a temporary object is created (7.4, 15.2). An object occupies a region of storage in its period of construction (15.7), throughout its lifetime (6.8), and in its period of destruction (15.7). [*Note:* A function is not an object, regardless of whether or not it occupies storage in the way that objects do. — *end note*] The properties of an object are determined when the object is created. An object can have a name (Clause 6). An object has a storage duration (6.7) which influences its lifetime (6.8). An object has a type (6.9). Some objects are polymorphic (13.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the *expressions* (Clause 8) used to access them.

2 Objects can contain other objects, called *subobjects*. A subobject can be a *member subobject* (12.2), a *base class subobject* (Clause 13), or an array element. An object that is not a subobject of any other object is called a *complete object*. If an object is created in storage associated with a member subobject or array element *e* (which may or may not be within its lifetime), the created object is a subobject of *e*'s containing object if:

(2.1) — the lifetime of *e*'s containing object has begun and not ended, and

(2.2) — the storage for the new object exactly overlays the storage location associated with *e*, and

(2.3) — the new object is of the same type as *e* (ignoring cv-qualification).

[*Note:* If the subobject contains a reference member or a `const` subobject, the name of the original subobject cannot be used to access the new object (6.8). — *end note*] [*Example:*

```
struct X { const int n; };
union U { X x; float f; };
void tong() {
  U u = {{ 1 }};
  u.f = 5.f;                          // OK, creates new subobject of u (12.3)
  X *p = new (&u.x) X {2};            // OK, creates new subobject of u
  assert(p->n == 2);                  // OK
  assert(*std::launder(&u.x.n) == 2); // OK
  assert(u.x.n == 2);                 // undefined behavior, u.x does not name new subobject
```

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

```
}
```

— *end example*]

3 If a complete object is created (8.3.4) in storage associated with another object e of type "array of N `unsigned char`" or of type "array of N `std::byte`" (21.2.1), that array *provides storage* for the created object if:

(3.1) — the lifetime of e has begun and not ended, and

(3.2) — the storage for the new object fits entirely within e, and

(3.3) — there is no smaller array object that satisfies these constraints.

[*Note:* If that portion of the array previously provided storage for another object, the lifetime of that object ends because its storage was reused (6.8). — *end note*] [*Example:*

```
template<typename ...T>
struct AlignedUnion {
  alignas(T...) unsigned char data[max(sizeof(T)...)];
};
int f() {
  AlignedUnion<int, char> au;
  int *p = new (au.data) int;      // OK, au.data provides storage
  char *c = new (au.data) char(); // OK, ends lifetime of *p
  char *d = new (au.data + 1) char();
  return *c + *d; // OK
}

struct A { unsigned char a[32]; };
struct B { unsigned char b[16]; };
A a;
B *b = new (a.a + 8) B;       // a.a provides storage for *b
int *p = new (b->b + 4) int; // b->b provides storage for *p
                              // a.a does not provide storage for *p (directly),
                              // but *p is nested within a (see below)
```

— *end example*]

4 An object a is *nested within* another object b if:

(4.1) — a is a subobject of b, or

(4.2) — b provides storage for a, or

(4.3) — there exists an object c where a is nested within c, and c is nested within b.

5 For every object `x`, there is some object called the *complete object of* `x`, determined as follows:

(5.1) — If `x` is a complete object, then the complete object of `x` is itself.

(5.2) — Otherwise, the complete object of `x` is the complete object of the (unique) object that contains `x`.

6 If a complete object, a data member (12.2), or an array element is of class type, its type is considered the *most derived class*, to distinguish it from the class type of any base class subobject; an object of a most derived class type or of a non-class type is called a *most derived object*.

7 Unless it is a bit-field (12.2.4), a most derived object shall have a nonzero size and shall occupy one or more bytes of storage. Base class subobjects may have zero size. An object of trivially copyable or standard-layout type (6.9) shall occupy contiguous bytes of storage.

8 Unless an object is a bit-field or a base class subobject of zero size, the address of that object is the address of the first byte it occupies. Two objects a and b with overlapping lifetimes that are not bit-fields may have the same address if one is nested within the other, or if at least one is a base class subobject of zero size and

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

they are of different types; otherwise, they have distinct addresses.[5]

[*Example:*

```
static const char test1 = 'x';
static const char test2 = 'x';
const bool b = &test1 != &test2;        // always true
```

— *end example*]

9 [*Note:* C++ provides a variety of fundamental types and several ways of composing new types from existing types (6.9). — *end note*]

4.6 Program execution [intro.execution]

1 The semantic descriptions in this document define a parameterized nondeterministic abstract machine. This document places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.[6]

2 Certain aspects and operations of the abstract machine are described in this document as implementation-defined (for example, `sizeof(int)`). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects.[7] Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the "corresponding instance" below).

3 Certain other aspects and operations of the abstract machine are described in this document as unspecified (for example, evaluation of expressions in a *new-initializer* if the allocation function fails to allocate memory (8.3.4)). Where possible, this document defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution for a given program and a given input.

4 Certain other operations are described in this document as undefined (for example, the effect of attempting to modify a `const` object). [*Note:* This document imposes no requirements on the behavior of programs that contain undefined behavior. — *end note*]

5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

6 An instance of each object with automatic storage duration (6.7.3) 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).

7 The least requirements on a conforming implementation are:

(7.1) — Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine.

(7.2) — At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.

5) Under the "as-if" rule an implementation is allowed to store two objects at the same machine address or not store an object at all if the program cannot observe the difference (4.6).

6) This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this document as long as the result is *as if* the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.

7) This documentation also includes conditionally-supported constructs and locale-specific behavior. See 4.1.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

(7.3) — The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.

These collectively are referred to as the *observable behavior* of the program. [*Note:* More stringent correspondences between abstract and actual semantics may be defined by each implementation. — *end note*]

8 [*Note:* Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative.[8] For example, 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. — *end note*]

9 A *constituent expression* is defined as follows:

(9.1) — The constituent expression of an expression is that expression.

(9.2) — The constituent expressions of a *braced-init-list* or of a (possibly parenthesized) *expression-list* are the constituent expressions of the elements of the respective list.

(9.3) — The constituent expressions of a *brace-or-equal-initializer* of the form `=` *initializer-clause* are the constituent expressions of the *initializer-clause*.

[*Example:*

```
struct A { int x; };
struct B { int y; struct A a; };
B b = { 5, { 1+1 } };
```

The constituent expressions of the *initializer* used for the initialization of `b` are `5` and `1+1`. — *end example*]

10 The *immediate subexpressions* of an expression `e` are

(10.1) — the constituent expressions of `e`'s operands (Clause 8),

(10.2) — any function call that `e` implicitly invokes,

(10.3) — if `e` is a *lambda-expression* (8.1.5), the initialization of the entities captured by copy and the constituent

8) Overloaded operators are never assumed to be associative or commutative.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

expressions of the *initializer* of the *init-captures*,

(10.4) — if `e` is a function call (8.2.2) or implicitly invokes a function, the constituent expressions of each default argument (11.3.6) used in the call, or

(10.5) — if `e` creates an aggregate object (11.6.1), the constituent expressions of each default member initializer (12.2) used in the initialization.

11 A *subexpression* of an expression `e` is an immediate subexpression of `e` or a subexpression of an immediate subexpression of `e`. [*Note:* Expressions appearing in the *compound-statement* of a *lambda-expression* are not subexpressions of the *lambda-expression*. — *end note*]

12 A *full-expression* is

(12.1) — an unevaluated operand (Clause 8),

(12.2) — a *constant-expression* (8.20),

(12.3) — an *init-declarator* (Clause 11) or a *mem-initializer* (15.6.2), including the constituent expressions of the initializer,

(12.4) — an invocation of a destructor generated at the end of the lifetime of an object other than a temporary object (15.2), or

(12.5) — an expression that is not a subexpression of another expression and that is not otherwise part of a full-expression.

If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression. For an initializer, performing the initialization of the entity (including evaluating default member initializers of an aggregate) is also considered part of the full-expression. [*Example:*

```
struct S {
  S(int i): I(i) { }        // full-expression is initialization of I
  int& v() { return I; }
  ~S() noexcept(false) { }
private:
  int I;
};

S s1(1);                    // full-expression is call of S::S(int)
void f() {
  S s2 = 2;                 // full-expression is call of S::S(int)
  if (S(3).v())             // full-expression includes lvalue-to-rvalue and
                            // int to bool conversions, performed before
                            // temporary is deleted at end of full-expression
  { }
  bool b = noexcept(S());   // exception specification of destructor of S
                            // considered for noexcept
  // full-expression is destruction of s2 at end of block
}
struct B {
      B(S = S(0));
   };
   B b[2] = { B(), B() };   // full-expression is the entire initialization
                            // including the destruction of temporaries
```

— *end example*]

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

13 [*Note:* The evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. For example, subexpressions involved in evaluating default arguments (11.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — *end note*]

14 Reading an object designated by a `volatile` glvalue (6.10), modifying an object, calling a library I/O function, 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 (or a subexpression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access through a volatile glvalue is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the `volatile` access may not have completed yet.

15 *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (4.7), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B (or, equivalently, B is *sequenced after* A), then the execution of A shall precede the execution of B. If A is not sequenced before B and B is not sequenced before A, then A and B are *unsequenced*. [*Note:* The execution of unsequenced evaluations can overlap. — *end note*] Evaluations A and B are *indeterminately sequenced* when either A is sequenced before B or B is sequenced before A, but it is unspecified which. [*Note:* Indeterminately sequenced evaluations cannot overlap, but either could be executed first. — *end note*] An expression X is said to be sequenced before an expression Y if every value computation and every side effect associated with the expression X is sequenced before every value computation and every side effect associated with the expression Y.

16 Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.[9]

17 Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. [*Note:* In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. — *end note*] The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a memory location (4.4) is unsequenced relative to either another side effect on the same memory location or a value computation using the value of any object in the same memory location, and they are not potentially concurrent (4.7), the behavior is undefined. [*Note:* The next section imposes similar, but more complex restrictions on potentially concurrent computations. — *end note*]

[*Example:*

```
void g(int i) {
  i = 7, i++, i++;     // i becomes 9

  i = i++ + 1;         // the value of i is incremented
  i = i++ + i;         // the behavior is undefined
  i = i + 1;           // the value of i is incremented
}
```

— *end example*]

18 When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. For each function invocation F, for every evaluation A that occurs within F and every evaluation B that does not

9) As specified in 15.2, after a full-expression is evaluated, a sequence of zero or more invocations of destructor functions for temporary objects takes place, usually in reverse order of the construction of each temporary object.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

occur within F but is evaluated on the same thread and as part of the same signal handler (if any), either A is sequenced before B or B is sequenced before A.[10] [*Note:* If A and B would not otherwise be sequenced then they are indeterminately sequenced. — *end note*] Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. [*Example:* Evaluation of a *new-expression* invokes one or more allocation and constructor functions; see 8.3.4. For another example, invocation of a conversion function (15.3.2) can arise in contexts in which no function call syntax appears. — *end example*] The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

19 If a signal handler is executed as a result of a call to the `std::raise` function, then the execution of the handler is sequenced after the invocation of the `std::raise` function and before its return. [*Note:* When a signal is received for another reason, the execution of the signal handler is usually unsequenced with respect to the rest of the program. — *end note*]

4.7 Multi-threaded executions and data races [intro.multithread]

1 A *thread of execution* (also known as a *thread*) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread. [*Note:* When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. — *end note*] Every thread in a program can potentially access every object and function in a program.[11] Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each thread proceeds as defined by the remainder of this document. The execution of the entire program consists of an execution of all of its threads. [*Note:* Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — *end note*] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

2 For a signal handler that is not executed as a result of a call to the `std::raise` function, it is unspecified which thread of execution contains the signal handler invocation.

4.7.1 Data races [intro.races]

1 The value of an object visible to a thread T at a particular point is the initial value of the object, a value assigned to the object by T, or a value assigned to the object by another thread, according to the rules below. [*Note:* In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — *end note*]

2 Two expression evaluations *conflict* if one of them modifies a memory location (4.4) and the other one reads or modifies the same memory location.

3 The library defines a number of atomic operations (Clause 32) and operations on mutexes (Clause 33) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [*Note:* For example, a call that acquires a mutex will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same mutex will perform a

10) In other words, function executions do not interleave with each other.

11) An object with automatic or thread storage duration (6.7) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (6.9.2).

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on A. "Relaxed" atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. — *end note*]

4 All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M. [*Note:* There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads may observe modifications to different objects in inconsistent orders. — *end note*]

5 A *release sequence* headed by a release operation A on an atomic object M is a maximal contiguous subsequence of side effects in the modification order of M, where the first operation is `A`, and every subsequent operation

(5.1) — is performed by the same thread that performed `A`, or

(5.2) — is an atomic read-modify-write operation.

6 Certain library calls *synchronize with* other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store (32.4). [*Note:* Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. — *end note*] [*Note:* The specifications of the synchronization operations define when one reads the value written by another. For atomic objects, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition "reads the value written" by the last mutex release. — *end note*]

7 An evaluation A *carries a dependency* to an evaluation B if

(7.1) — the value of A is used as an operand of B, unless:

(7.1.1) — B is an invocation of any specialization of `std::kill_dependency` (32.4), or

(7.1.2) — A is the left operand of a built-in logical AND (`&&`, see 8.14) or logical OR (`||`, see 8.15) operator, or

(7.1.3) — A is the left operand of a conditional (`?:`, see 8.16) operator, or

(7.1.4) — A is the left operand of the built-in comma (`,`) operator (8.19);

or

(7.2) — A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or

(7.3) — for some evaluation X, A carries a dependency to X, and X carries a dependency to B.

[*Note:* "Carries a dependency to" is a subset of "is sequenced before", and is similarly strictly intra-thread. — *end note*]

8 An evaluation A is *dependency-ordered before* an evaluation B if

(8.1) — A performs a release operation on an atomic object M, and, in another thread, B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or

(8.2) — for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

[*Note:* The relation "is dependency-ordered before" is analogous to "synchronizes with", but uses release/-consume in place of release/acquire. — *end note*]

9 An evaluation A *inter-thread happens before* an evaluation B if

(9.1) — A synchronizes with B, or

(9.2) — A is dependency-ordered before B, or

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

(9.3) — for some evaluation X

(9.3.1) — A synchronizes with X and X is sequenced before B, or

(9.3.2) — A is sequenced before X and X inter-thread happens before B, or

(9.3.3) — A inter-thread happens before X and X inter-thread happens before B.

[*Note:* The "inter-thread happens before" relation describes arbitrary concatenations of "sequenced before", "synchronizes with" and "dependency-ordered before" relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with "dependency-ordered before" followed by "sequenced before". The reason for this limitation is that a consume operation participating in a "dependency-ordered before" relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of "sequenced before". The reasons for this limitation are (1) to permit "inter-thread happens before" to be transitively closed and (2) the "happens before" relation, defined below, provides for relationships consisting entirely of "sequenced before". — *end note*]

10 An evaluation A *happens before* an evaluation B (or, equivalently, B *happens after* A) if:

(10.1) — A is sequenced before B, or

(10.2) — A inter-thread happens before B.

The implementation shall ensure that no program execution demonstrates a cycle in the "happens before" relation. [*Note:* This cycle would otherwise be possible only through the use of consume operations. — *end note*]

11 An evaluation A *strongly happens before* an evaluation B if either

(11.1) — A is sequenced before B, or

(11.2) — A synchronizes with B, or

(11.3) — A strongly happens before X and X strongly happens before B.

[*Note:* In the absence of consume operations, the happens before and strongly happens before relations are identical. Strongly happens before essentially excludes consume operations. — *end note*]

12 A *visible side effect* A on a scalar object or bit-field M with respect to a value computation B of M satisfies the conditions:

(12.1) — A happens before B and

(12.2) — there is no other side effect X to M such that A happens before X and X happens before B.

The value of a non-atomic scalar object or bit-field M, as determined by evaluation B, shall be the value stored by the visible side effect A. [*Note:* If there is ambiguity about which side effect to a non-atomic object or bit-field is visible, then the behavior is either unspecified or undefined. — *end note*] [*Note:* This states that operations on ordinary objects are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined below, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution. — *end note*]

13 The value of an atomic object M, as determined by evaluation B, shall be the value stored by some side effect A that modifies M, where B does not happen before A. [*Note:* The set of such side effects is also restricted by the rest of the rules described here, and in particular, by the coherence requirements below. — *end note*]

14 If an operation A that modifies an atomic object M happens before an operation B that modifies M, then A shall be earlier than B in the modification order of M. [*Note:* This requirement is known as write-write coherence. — *end note*]

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

15 If a value computation A of an atomic object M happens before a value computation B of M, and A takes its value from a side effect X on M, then the value computed by B shall either be the value stored by X or the value stored by a side effect Y on M, where Y follows X in the modification order of M. [*Note:* This requirement is known as read-read coherence. — *end note*]

16 If a value computation A of an atomic object M happens before an operation B that modifies M, then A shall take its value from a side effect X on M, where X precedes B in the modification order of M. [*Note:* This requirement is known as read-write coherence. — *end note*]

17 If a side effect X on an atomic object M happens before a value computation B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M. [*Note:* This requirement is known as write-read coherence. — *end note*]

18 [*Note:* The four preceding coherence requirements effectively disallow compiler reordering of atomic operations to a single object, even if both operations are relaxed loads. This effectively makes the cache coherence guarantee provided by most hardware available to C++ atomic operations. — *end note*]

19 [*Note:* The value observed by a load of an atomic depends on the "happens before" relation, which depends on the values observed by loads of atomics. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the "happens before" relation derived as described above, satisfy the resulting constraints as imposed here. — *end note*]

20 Two actions are *potentially concurrent* if

(20.1) — they are performed by different threads, or

(20.2) — they are unsequenced, at least one is performed by a signal handler, and they are not both performed by the same signal handler invocation.

The execution of a program contains a *data race* if it contains two potentially concurrent conflicting actions, at least one of which is not atomic, and neither happens before the other, except for the special case for signal handlers described below. Any such data race results in undefined behavior. [*Note:* It can be shown that programs that correctly use mutexes and `memory_order_seq_cst` operations to prevent all data races and use no other synchronization operations behave as if the operations executed by their constituent threads were simply interleaved, with each value computation of an object being taken from the last side effect on that object in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to data-race-free programs, and data-race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. — *end note*]

21 Two accesses to the same object of type `volatile std::sig_atomic_t` do not result in a data race if both occur in the same thread, even if one or more occurs in a signal handler. For each signal handler invocation, evaluations performed by the thread invoking a signal handler can be divided into two groups A and B, such that no evaluations in B happen before evaluations in A, and the evaluations of such `volatile std::sig_atomic_t` objects take values as though all evaluations in A happened before the execution of the signal handler and the execution of the signal handler happened before all evaluations in B.

22 [*Note:* Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this document, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. Reordering of atomic loads in cases in which the atomics in question may alias is also generally precluded, since this may violate the coherence rules. — *end note*]

23 [*Note:* Transformations that introduce a speculative read of a potentially shared memory location may not

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

preserve the semantics of the C++ program as defined in this document, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection. — *end note*]

4.7.2 Forward progress [intro.progress]

1 The implementation may assume that any thread will eventually do one of the following:

(1.1) — terminate,

(1.2) — make a call to a library I/O function,

(1.3) — perform an access through a volatile glvalue, or

(1.4) — perform a synchronization operation or an atomic operation.

[*Note:* This is intended to allow compiler transformations such as removal of empty loops, even when termination cannot be proven. — *end note*]

2 Executions of atomic functions that are either defined to be lock-free (32.8) or indicated as lock-free (32.5) are *lock-free executions*.

(2.1) — If there is only one thread that is not blocked (3.6) in a standard library function, a lock-free execution in that thread shall complete. [*Note:* Concurrently executing threads may prevent progress of a lock-free execution. For example, this situation can occur with load-locked store-conditional implementations. This property is sometimes termed obstruction-free. — *end note*]

(2.2) — When one or more lock-free executions run concurrently, at least one should complete. [*Note:* It is difficult for some implementations to provide absolute guarantees to this effect, since repeated and particularly inopportune interference from other threads may prevent forward progress, e.g., by repeatedly stealing a cache line for unrelated purposes between load-locked and store-conditional instructions. Implementations should ensure that such effects cannot indefinitely delay progress under expected operating conditions, and that such anomalies can therefore safely be ignored by programmers. Outside this document, this property is sometimes termed lock-free. — *end note*]

3 During the execution of a thread of execution, each of the following is termed an *execution step*:

(3.1) — termination of the thread of execution,

(3.2) — performing an access through a volatile glvalue, or

(3.3) — completion of a call to a library I/O function, a synchronization operation, or an atomic operation.

4 An invocation of a standard library function that blocks (3.6) is considered to continuously execute execution steps while waiting for the condition that it blocks on to be satisfied. [*Example:* A library I/O function that blocks until the I/O operation is complete can be considered to continuously check whether the operation is complete. Each such check might consist of one or more execution steps, for example using observable behavior of the abstract machine. — *end example*]

5 [*Note:* Because of this and the preceding requirement regarding what threads of execution have to perform eventually, it follows that no thread of execution can execute forever without an execution step occurring. — *end note*]

6 A thread of execution *makes progress* when an execution step occurs or a lock-free execution does not complete because there are other concurrent threads that are not blocked in a standard library function (see above).

7 For a thread of execution providing *concurrent forward progress guarantees*, the implementation ensures that the thread will eventually make progress for as long as it has not terminated. [*Note:* This is required regardless of whether or not other threads of executions (if any) have been or are making progress. To eventually fulfill this requirement means that this will happen in an unspecified but finite amount of time. — *end note*]

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

8 It is implementation-defined whether the implementation-created thread of execution that executes `main` (6.6.1) and the threads of execution created by `std::thread` (33.3.2) provide concurrent forward progress guarantees. [*Note:* General-purpose implementations should provide these guarantees. — *end note*]

9 For a thread of execution providing *parallel forward progress guarantees*, the implementation is not required to ensure that the thread will eventually make progress if it has not yet executed any execution step; once this thread has executed a step, it provides concurrent forward progress guarantees.

10 [*Note:* This does not specify a requirement for when to start this thread of execution, which will typically be specified by the entity that creates this thread of execution. For example, a thread of execution that provides concurrent forward progress guarantees and executes tasks from a set of tasks in an arbitrary order, one after the other, satisfies the requirements of parallel forward progress for these tasks. — *end note*]

11 For a thread of execution providing *weakly parallel forward progress guarantees*, the implementation does not ensure that the thread will eventually make progress.

12 [*Note:* Threads of execution providing weakly parallel forward progress guarantees cannot be expected to make progress regardless of whether other threads make progress or not; however, blocking with forward progress guarantee delegation, as defined below, can be used to ensure that such threads of execution make progress eventually. — *end note*]

13 Concurrent forward progress guarantees are stronger than parallel forward progress guarantees, which in turn are stronger than weakly parallel forward progress guarantees. [*Note:* For example, some kinds of synchronization between threads of execution may only make progress if the respective threads of execution provide parallel forward progress guarantees, but will fail to make progress under weakly parallel guarantees. — *end note*]

14 When a thread of execution P is specified to *block with forward progress guarantee delegation* on the completion of a set S of threads of execution, then throughout the whole time of P being blocked on S, the implementation shall ensure that the forward progress guarantees provided by at least one thread of execution in S is at least as strong as P's forward progress guarantees. [*Note:* It is unspecified which thread or threads of execution in S are chosen and for which number of execution steps. The strengthening is not permanent and not necessarily in place for the rest of the lifetime of the affected thread of execution. As long as P is blocked, the implementation has to eventually select and potentially strengthen a thread of execution in S. — *end note*] Once a thread of execution in S terminates, it is removed from S. Once S is empty, P is unblocked.

15 [*Note:* A thread of execution B thus can temporarily provide an effectively stronger forward progress guarantee for a certain amount of time, due to a second thread of execution A being blocked on it with forward progress guarantee delegation. In turn, if B then blocks with forward progress guarantee delegation on C, this may also temporarily provide a stronger forward progress guarantee to C. — *end note*]

16 [*Note:* If all threads of execution in S finish executing (e.g., they terminate and do not use blocking synchronization incorrectly), then P's execution of the operation that blocks with forward progress guarantee delegation will not result in P's progress guarantee being effectively weakened. — *end note*]

17 [*Note:* This does not remove any constraints regarding blocking synchronization for threads of execution providing parallel or weakly parallel forward progress guarantees because the implementation is not required to strengthen a particular thread of execution whose too-weak progress guarantee is preventing overall progress. — *end note*]

18 An implementation should ensure that the last value (in modification order) assigned by an atomic or synchronization operation will become visible to all other threads in a finite period of time.

4.8 Acknowledgments [intro.ack]

1 The C++ programming language as described in this document is based on the language as described in Chapter R (Reference Manual) of Stroustrup: *The C++ Programming Language* (second edition, Addison-Wesley Publishing Company, ISBN 0-201-53992-6, copyright ©1991 AT&T). That, in turn, is based on the C

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

programming language as described in Appendix A of Kernighan and Ritchie: *The C Programming Language* (Prentice-Hall, 1978, ISBN 0-13-110163-3, copyright ©1978 AT&T).

2 Portions of the library Clauses of this document are based on work by P.J. Plauger, which was published as *The Draft Standard C++ Library* (Prentice-Hall, ISBN 0-13-117003-1, copyright ©1995 P.J. Plauger).

3 POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.

4 ECMAScript® is a registered trademark of Ecma International.

5 All rights in these originals are reserved.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

5 Lexical conventions [lex]

5.1 Separate translation [lex.separate]

1 The text of the program is kept in units called *source files* in this document. A source file together with all the headers (20.5.1.2) and source files included (19.2) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (19.1) preprocessing directives, is called a *translation unit*. [*Note:* A C++ program need not all be translated at the same time. — *end note*]

2 [*Note:* Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (6.5) 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 can be separately translated and then later linked to produce an executable program (6.5). — *end note*]

5.2 Phases of translation [lex.phases]

1 The precedence among the syntax rules of translation is specified by the following phases.[12]

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary. The set of physical source file characters accepted is implementation-defined. Any source file character not in the basic source character set (5.3) is replaced by the *universal-character-name* that designates that character. An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a *universal-character-name* (e.g., using the `\uXXXX` notation), are handled equivalently except where this replacement is reverted (5.4) in a raw string literal.
2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. Except for splices reverted in a raw string literal, if a splice results in a character sequence that matches the syntax of a *universal-character-name*, the behavior is undefined. A source file that is not empty and that does not end in a new-line character, or that ends in a new-line character immediately preceded by a backslash character before any such splicing takes place, shall be processed as if an additional new-line character were appended to the file.
3. The source file is decomposed into preprocessing tokens (5.4) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.[13] 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 unspecified. The process of dividing a source file's characters into preprocessing tokens is context-dependent. [*Example:* See the handling of `<` within a `#include` preprocessing directive. — *end example*]
4. Preprocessing directives are executed, macro invocations are expanded, and `_Pragma` unary operator expressions are executed. If a character sequence that matches the syntax of a *universal-character-name* is produced by token concatenation (19.3.3), the behavior is undefined. A `#include` preprocessing

12) Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

13) A partial preprocessing token would arise from a source file ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a *header-name* that is missing the closing `"` or `>`. A partial comment would arise from a source file ending with an unclosed `/*` comment.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

5. Each source character set member in a character literal or a string literal, as well as each escape sequence and *universal-character-name* in a character literal or a non-raw string literal, is converted to the corresponding member of the execution character set (5.13.3, 5.13.5); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[14]
6. Adjacent string literal tokens are concatenated.
7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token (5.6). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [*Note:* The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (17.2). — *end note*] [*Note:* Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. — *end note*]
8. Translated translation units and instantiation units are combined as follows: [*Note:* Some or all of these may be supplied from a library. — *end note*] Each translated translation unit is examined to produce a list of required instantiations. [*Note:* This may include instantiations which have been explicitly requested (17.7.2). — *end note*] The definitions of the required templates are located. It is implementation-defined whether the source of the translation units containing these definitions is required to be available. [*Note:* An implementation could encode sufficient information into the translated translation unit so as to ensure the source is not required here. — *end note*] All the required instantiations are performed to produce *instantiation units*. [*Note:* These are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions. — *end note*] The program is ill-formed if any instantiation fails.
9. All external entity references are resolved. Library components are linked to satisfy external references to entities 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.

5.3 Character sets [lex.charset]

1 The *basic source character set* consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:[15]

```
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
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

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

hex-quad:
  hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:
  `\u` *hex-quad*
  `\U` *hex-quad hex-quad*

The character designated by the *universal-character-name* `\UNNNNNNNN` is that character whose character short name in ISO/IEC 10646 is `NNNNNNNN`; the character designated by the *universal-character-name* `\uNNNN`

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

15) The glyphs for the members of the basic source character set are intended to identify characters from the subset of ISO/IEC 10646 which corresponds to the ASCII character set. However, because the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, an implementation is required to document how the basic source characters are represented in source files.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

is that character whose character short name in ISO/IEC 10646 is `0000NNNN`. If the hexadecimal value for a *universal-character-name* corresponds to a surrogate code point (in the range 0xD800–0xDFFF, inclusive), the program is ill-formed. Additionally, if the hexadecimal value for a *universal-character-name* outside the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of a character or string literal corresponds to a control character (in either of the ranges 0x00–0x1F or 0x7F–0x9F, both inclusive) or to a character in the basic source character set, the program is ill-formed.[16]

3 The *basic execution character set* and the *basic execution wide-character set* shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a *null character* (respectively, *null wide character*), whose value is 0. For each basic execution character set, the values of the members shall be non-negative and distinct from one another. 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. The *execution character set* and the *execution wide-character set* are implementation-defined supersets of the basic execution character set and the basic execution wide-character set, respectively. The values of the members of the execution character sets and the sets of additional members are locale-specific.

5.4 Preprocessing tokens [lex.pptoken]

preprocessing-token:
- *header-name*
- *identifier*
- *pp-number*
- *character-literal*
- *user-defined-character-literal*
- *string-literal*
- *user-defined-string-literal*
- *preprocessing-op-or-punc*
- each non-white-space character that cannot be one of the above

1 Each preprocessing token that is converted to a token (5.6) shall have the lexical form of a keyword, an identifier, a literal, an operator, or a punctuator.

2 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 literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and 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 (5.7), or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in Clause 19, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

3 If the input stream has been parsed into preprocessing tokens up to a given character:

(3.1) — If the next character begins a sequence of characters that could be the prefix and initial double quote of a raw string literal, such as `R"`, the next preprocessing token shall be a raw string literal. Between the initial and final double quote characters of the raw string, any transformations performed in phases 1 and 2 (*universal-character-name*s and line splicing) are reverted; this reversion shall apply before any *d-char*, *r-char*, or delimiting parenthesis is identified. The raw string literal is defined as the shortest sequence of characters that matches the raw-string pattern

encoding-prefix$_{opt}$ `R` *raw-string*

16) A sequence of characters resembling a *universal-character-name* in an *r-char-sequence* (5.13.5) does not form a *universal-character-name*.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

(3.2) — Otherwise, if the next three characters are `<::` and the subsequent character is neither `:` nor `>`, the `<` is treated as a preprocessing token by itself and not as the first character of the alternative token `<:`.

(3.3) — Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail, except that a *header-name* (5.8) is only formed within a `#include` directive (19.2).

[*Example:*

```
#define R "x"
const char* s = R"y";          // ill-formed raw string, not "x" "y"
```

— *end example*]

4 [*Example:* The program fragment `0xe+foo` is parsed as a preprocessing number token (one that is not a valid floating or integer literal token), even though a parse as three preprocessing tokens `0xe`, `+`, and `foo` might produce a valid expression (for example, if `foo` were a macro defined as `1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating literal token), whether or not `E` is a macro name. — *end example*]

5 [*Example:* The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which, if `x` and `y` have integral types, violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression. — *end example*]

5.5 Alternative tokens [lex.digraph]

1 Alternative token representations are provided for some operators and punctuators.[17]

2 In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.[18] The set of alternative tokens is defined in Table 1.

Table 1 — Alternative tokens

Alternative	Primary	Alternative	Primary	Alternative	Primary
`<%`	`{`	`and`	`&&`	`and_eq`	`&=`
`%>`	`}`	`bitor`	`|`	`or_eq`	`|=`
`<:`	`[`	`or`	`||`	`xor_eq`	`^=`
`:>`	`]`	`xor`	`^`	`not`	`!`
`%:`	`#`	`compl`	`~`	`not_eq`	`!=`
`%:%:`	`##`	`bitand`	`&`		

5.6 Tokens [lex.token]

token:
- *identifier*
- *keyword*
- *literal*
- *operator*
- *punctuator*

1 There are five kinds of tokens: identifiers, keywords, literals,[19] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "white space"), as described

17) These include "digraphs" and additional reserved words. The term "digraph" (token consisting of two characters) is not perfectly descriptive, since one of the alternative preprocessing-tokens is `%:%:` and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs".

18) Thus the "stringized" values (19.3.2) of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

19) Literals include strings and character and numeric literals.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

below, are ignored except as they serve to separate tokens. [*Note:* Some white space is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. — *end note*]

5.7 Comments [lex.comment]

1 The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates immediately before the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required. [*Note:* The comment characters `//`, `/*`, and `*/` have no special meaning within a `//` comment and are treated just like other characters. Similarly, the comment characters `//` and `/*` have no special meaning within a `/*` comment. — *end note*]

5.8 Header names [lex.header]

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 new-line and `>`

q-char-sequence:
  q-char
  q-char-sequence q-char

q-char:
  any member of the source character set except new-line and `"`

1 [*Note:* Header name preprocessing tokens only appear within a `#include` preprocessing directive (see 5.4). — *end note*] The sequences in both forms of *header-name*s are mapped in an implementation-defined manner to headers or to external source file names as specified in 19.2.

2 The appearance of either of the characters `'` or `\` or of either of the character sequences `/*` or `//` in a *q-char-sequence* or an *h-char-sequence* is conditionally-supported with implementation-defined semantics, as is the appearance of the character `"` in an *h-char-sequence*.[20]

5.9 Preprocessing numbers [lex.ppnumber]

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

20) Thus, a sequence of characters that resembles an escape sequence might result in an error, be interpreted as the character corresponding to the escape sequence, or have a completely different meaning, depending on the implementation.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

1 Preprocessing number tokens lexically include all integer literal tokens (5.13.2) and all floating literal tokens (5.13.4).

2 A preprocessing number does not have a type or a value; it acquires both after a successful conversion to an integer literal token or a floating literal token.

5.10 Identifiers [lex.name]

identifier:
  identifier-nondigit
  identifier identifier-nondigit
  identifier digit

identifier-nondigit:
  nondigit
  universal-character-name

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`

1 An identifier is an arbitrarily long sequence of letters and digits. Each *universal-character-name* in an identifier shall designate a character whose encoding in ISO 10646 falls into one of the ranges specified in Table 2. The initial element shall not be a *universal-character-name* designating a character whose encoding falls into one of the ranges specified in Table 3. Upper- and lower-case letters are different. All characters are significant.[21]

Table 2 — Ranges of characters allowed

00A8	00AA	00AD	00AF	00B2-00B5
00B7-00BA	00BC-00BE	00C0-00D6	00D8-00F6	00F8-00FF
0100-167F	1681-180D	180F-1FFF		
200B-200D	202A-202E	203F-2040	2054	2060-206F
2070-218F	2460-24FF	2776-2793	2C00-2DFF	2E80-2FFF
3004-3007	3021-302F	3031-D7FF		
F900-FD3D	FD40-FDCF	FDF0-FE44	FE47-FFFD	
10000-1FFFD	20000-2FFFD	30000-3FFFD	40000-4FFFD	50000-5FFFD
60000-6FFFD	70000-7FFFD	80000-8FFFD	90000-9FFFD	A0000-AFFFD
B0000-BFFFD	C0000-CFFFD	D0000-DFFFD	E0000-EFFFD	

Table 3 — Ranges of characters disallowed initially (combining characters)

0300-036F	1DC0-1DFF	20D0-20FF	FE20-FE2F

21) On systems in which linkers cannot accept extended characters, an encoding of the *universal-character-name* may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the `\u` in a *universal-character-name*. Extended characters may produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers. In C++, upper- and lower-case letters are considered different for all identifiers, including external identifiers.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

2 The identifiers in Table 4 have a special meaning when appearing in a certain context. When referred to in the grammar, these identifiers are used explicitly rather than using the *identifier* grammar production. Unless otherwise specified, any ambiguity as to whether a given *identifier* has a special meaning is resolved to interpret the token as a regular *identifier*.

Table 4 — Identifiers with special meaning

`override`	`final`

3 In addition, some identifiers are reserved for use by C++ implementations and shall not be used otherwise; no diagnostic is required.

(3.1) — Each identifier that contains a double underscore `__` or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use.

(3.2) — Each identifier that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

5.11 Keywords [lex.key]

1 The identifiers shown in Table 5 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7) except in an *attribute-token* (10.6.1):

Table 5 — Keywords

`alignas`	`continue`	`friend`	`register`	`true`
`alignof`	`decltype`	`goto`	`reinterpret_cast`	`try`
`asm`	`default`	`if`	`return`	`typedef`
`auto`	`delete`	`inline`	`short`	`typeid`
`bool`	`do`	`int`	`signed`	`typename`
`break`	`double`	`long`	`sizeof`	`union`
`case`	`dynamic_cast`	`mutable`	`static`	`unsigned`
`catch`	`else`	`namespace`	`static_assert`	`using`
`char`	`enum`	`new`	`static_cast`	`virtual`
`char16_t`	`explicit`	`noexcept`	`struct`	`void`
`char32_t`	`export`	`nullptr`	`switch`	`volatile`
`class`	`extern`	`operator`	`template`	`wchar_t`
`const`	`false`	`private`	`this`	`while`
`constexpr`	`float`	`protected`	`thread_local`	
`const_cast`	`for`	`public`	`throw`	

[*Note:* The `export` and `register` keywords are unused but are reserved for future use. — *end note*]

2 Furthermore, the alternative representations shown in Table 6 for certain operators and punctuators (5.5) are reserved and shall not be used otherwise:

Table 6 — Alternative representations

`and`	`and_eq`	`bitand`	`bitor`	`compl`	`not`
`not_eq`	`or`	`or_eq`	`xor`	`xor_eq`	

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

5.12 Operators and punctuators [lex.operators]

1 The lexical representation of C++ programs includes a number of preprocessing tokens which are used in the syntax of the preprocessor or are converted into tokens for operators and punctuators:

preprocessing-op-or-punc: one of

```
{     }         [         ]         #         ##        (         )
<:    :>        <%        %>        %:        %:%:      ;         :         ...
new   delete    ?         ::        .         .*
+     -         *         /         %         ^         &         |         ~
!     =         <         >         +=        -=        *=        /=        %=
^=    &=        |=        <<        >>        >>=       <<=       ==        !=
<=    >=        &&        ||        ++        --        ,         ->*       ->
and   and_eq    bitand    bitor     compl     not       not_eq
or    or_eq     xor       xor_eq
```

Each *preprocessing-op-or-punc* is converted to a single token in translation phase 7 (5.2).

5.13 Literals [lex.literal]

5.13.1 Kinds of literals [lex.literal.kinds]

1 There are several kinds of literals.[22]

literal:
- *integer-literal*
- *character-literal*
- *floating-literal*
- *string-literal*
- *boolean-literal*
- *pointer-literal*
- *user-defined-literal*

5.13.2 Integer literals [lex.icon]

integer-literal:
- *binary-literal integer-suffix*$_{opt}$
- *octal-literal integer-suffix*$_{opt}$
- *decimal-literal integer-suffix*$_{opt}$
- *hexadecimal-literal integer-suffix*$_{opt}$

binary-literal:
- `0b` *binary-digit*
- `0B` *binary-digit*
- *binary-literal* `'`$_{opt}$ *binary-digit*

octal-literal:
- `0`
- *octal-literal* `'`$_{opt}$ *octal-digit*

decimal-literal:
- *nonzero-digit*
- *decimal-literal* `'`$_{opt}$ *digit*

hexadecimal-literal:
- *hexadecimal-prefix hexadecimal-digit-sequence*

binary-digit:
- `0`
- `1`

22) The term "literal" generally designates, in this document, those tokens that are called "constants" in ISO C.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

octal-digit: one of
`0 1 2 3 4 5 6 7`

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

hexadecimal-prefix: one of
`0x 0X`

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence `'`$_{opt}$ *hexadecimal-digit*

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

unsigned-suffix: one of
`u U`

long-suffix: one of
`l L`

long-long-suffix: one of
`ll LL`

1 An *integer literal* is a sequence of digits that has no period or exponent part, with optional separating single quotes that are ignored when determining its value. An integer literal may have a prefix that specifies its base and a suffix that specifies its type. The lexically first digit of the sequence of digits is the most significant. A *binary integer literal* (base two) begins with `0b` or `0B` and consists of a sequence of binary digits. An *octal integer literal* (base eight) begins with the digit `0` and consists of a sequence of octal digits.[23] A *decimal integer literal* (base ten) begins with a digit other than `0` and consists of a sequence of decimal digits. A *hexadecimal integer literal* (base sixteen) begins with `0x` or `0X` and consists of a sequence of hexadecimal digits, which include the decimal digits and the letters `a` through `f` and `A` through `F` with decimal values ten through fifteen. [*Example:* The number twelve can be written `12`, `014`, `0XC`, or `0b1100`. The integer literals `1048576`, `1'048'576`, `0X100000`, `0x10'0000`, and `0'004'000'000` all have the same value. — *end example*]

2 The type of an integer literal is the first of the corresponding list in Table 7 in which its value can be represented.

3 If an integer literal cannot be represented by any type in its list and an extended integer type (6.9.1) can represent its value, it may have that extended integer type. If all of the types in the list for the integer literal are signed, the extended integer type shall be signed. If all of the types in the list for the integer literal are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. A program is ill-formed if one of its translation units contains an integer literal that cannot be represented by any of the allowed types.

23) The digits `8` and `9` are not octal digits.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

Table 7 — Types of integer literals

Suffix	Decimal literal	Binary, octal, or hexadecimal literal

none	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long long int`
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

5.13.3 Character literals [lex.ccon]

character-literal:
  encoding-prefix$_{opt}$ ' *c-char-sequence* '

encoding-prefix: one of
  `u8` `u` `U` `L`

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
  universal-character-name

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

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

1 A character literal is one or more characters enclosed in single quotes, as in `'x'`, optionally preceded by `u8`, `u`, `U`, or `L`, as in `u8'w'`, `u'x'`, `U'y'`, or `L'z'`, respectively.

2 A character literal that does not begin with `u8`, `u`, `U`, or `L` is an *ordinary character literal*. An ordinary character literal that contains a single *c-char* representable in the execution character set has type `char`, with value equal to the numerical value of the encoding of the *c-char* in the execution character set. An ordinary character literal that contains more than one *c-char* is a *multicharacter literal*. A multicharacter literal, or an ordinary character literal containing a single *c-char* not representable in the execution character set, is conditionally-supported, has type `int`, and has an implementation-defined value.

3 A character literal that begins with `u8`, such as `u8'w'`, is a character literal of type `char`, known as a *UTF-8 character literal*. The value of a UTF-8 character literal is equal to its ISO 10646 code point value, provided that the code point value is representable with a single UTF-8 code unit (that is, provided it is in the C0 Controls and Basic Latin Unicode block). If the value is not representable with a single UTF-8 code unit, the program is ill-formed. A UTF-8 character literal containing multiple *c-char*s is ill-formed.

4 A character literal that begins with the letter `u`, such as `u'x'`, is a character literal of type `char16_t`. The value of a `char16_t` character literal containing a single *c-char* is equal to its ISO 10646 code point value, provided that the code point is representable with a single 16-bit code unit. (That is, provided it is a basic multi-lingual plane code point.) If the value is not representable within 16 bits, the program is ill-formed. A `char16_t` character literal containing multiple *c-char*s is ill-formed.

5 A character literal that begins with the letter `U`, such as `U'y'`, is a character literal of type `char32_t`. The value of a `char32_t` character literal containing a single *c-char* is equal to its ISO 10646 code point value. A `char32_t` character literal containing multiple *c-char*s is ill-formed.

6 A character literal that begins with the letter `L`, such as `L'z'`, is a *wide-character literal*. A wide-character literal has type `wchar_t`.[24] The value of a wide-character literal containing a single *c-char* has value equal to the numerical value of the encoding of the *c-char* in the execution wide-character set, unless the *c-char* has no representation in the execution wide-character set, in which case the value is implementation-defined. [*Note:* The type `wchar_t` is able to represent all members of the execution wide-character set (see 6.9.1). — *end note*] The value of a wide-character literal containing multiple *c-char*s is implementation-defined.

7 Certain non-graphic characters, the single quote `'`, the double quote `"`, the question mark `?`,[25] and the backslash `\`, can be represented according to Table 8. The double quote `"` and the question mark `?`, can be represented as themselves or by the escape sequences `\"` and `\?` respectively, but the single quote `'` and the backslash `\` shall be represented by the escape sequences `\'` and `\\` respectively. Escape sequences in which the character following the backslash is not listed in Table 8 are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

8 The escape `\ooo` consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape `\xhhh` consists of the backslash followed by `x` followed by one or more hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for `char` (for character literals with no prefix) or `wchar_t` (for character literals prefixed by `L`). [*Note:* If the value of a character literal prefixed by `u`, `u8`, or `U` is outside the range defined for its type, the program is ill-formed. — *end note*]

9 A *universal-character-name* is translated to the encoding, in the appropriate execution character set, of the character named. If there is no such encoding, the *universal-character-name* is translated to an implementation-defined encoding. [*Note:* In translation phase 1, a *universal-character-name* is introduced whenever an actual

24) They are intended for character sets where a character does not fit into a single byte.
25) Using an escape sequence for a question mark is supported for compatibility with ISO C++ 2014 and ISO C.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

Table 8 — Escape sequences

new-line	NL(LF)	`\n`
horizontal tab	HT	`\t`
vertical tab	VT	`\v`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
alert	BEL	`\a`
backslash	\	`\\`
question mark	?	`\?`
single quote	'	`\'`
double quote	"	`\"`
octal number	*ooo*	`\ooo`
hex number	*hhh*	`\xhhh`

extended character is encountered in the source text. Therefore, all extended characters are described in terms of *universal-character-name*s. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained. — *end note*]

5.13.4 Floating literals [lex.fcon]

floating-literal:
  decimal-floating-literal
  hexadecimal-floating-literal

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

hexadecimal-floating-literal:
  hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part floating-suffix$_{opt}$
  hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part floating-suffix$_{opt}$

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

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

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

binary-exponent-part:
  `p` *sign*$_{opt}$ *digit-sequence*
  `P` *sign*$_{opt}$ *digit-sequence*

sign: one of
  `+ -`

digit-sequence:
  digit
  digit-sequence `'`$_{opt}$ *digit*

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

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

1 A floating literal consists of an optional prefix specifying a base, an integer part, a radix point, a fraction part, an `e`, `E`, `p` or `P`, an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits if there is no prefix, or hexadecimal (base sixteen) digits if the prefix is `0x` or `0X`. The floating literal is a *decimal floating literal* in the former case and a *hexadecimal floating literal* in the latter case. Optional separating single quotes in a *digit-sequence* or *hexadecimal-digit-sequence* are ignored when determining its value. [*Example:* The floating literals `1.602'176'565e-19` and `1.602176565e-19` have the same value. — *end example*] Either the integer part or the fraction part (not both) can be omitted. Either the radix point or the letter `e` or `E` and the exponent (not both) can be omitted from a decimal floating literal. The radix point (but not the exponent) can be omitted from a hexadecimal floating literal. The integer part, the optional radix point, and the optional fraction part, form the *significand* of the floating literal. In a decimal floating literal, the exponent, if present, indicates the power of 10 by which the significand is to be scaled. In a hexadecimal floating literal, the exponent indicates the power of 2 by which the significand is to be scaled. [*Example:* The floating literals `49.625` and `0xC.68p+2` have the same value. — *end example*] If the scaled value is in the range of representable values for its type, the result is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner. The type of a floating literal is `double` unless explicitly specified by a suffix. The suffixes `f` and `F` specify `float`, the suffixes `l` and `L` specify `long double`. If the scaled value is not in the range of representable values for its type, the program is ill-formed.

5.13.5 String literals [lex.string]

string-literal:
encoding-prefix$_{opt}$ `"` *s-char-sequence*$_{opt}$ `"`
encoding-prefix$_{opt}$ `R` *raw-string*

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
universal-character-name

raw-string:
`"` *d-char-sequence*$_{opt}$ `(` *r-char-sequence*$_{opt}$ `)` *d-char-sequence*$_{opt}$ `"`

r-char-sequence:
r-char
r-char-sequence r-char

r-char:
any member of the source character set, except
a right parenthesis `)` followed by the initial *d-char-sequence*
(which may be empty) followed by a double quote `"`.

d-char-sequence:
d-char
d-char-sequence d-char

d-char:
any member of the basic source character set except:
space, the left parenthesis `(`, the right parenthesis `)`, the backslash `\`,
and the control characters representing horizontal tab,
vertical tab, form feed, and newline.

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017

1 A *string-literal* is a sequence of characters (as defined in 5.13.3) surrounded by double quotes, optionally prefixed by `R`, `u8`, `u8R`, `u`, `uR`, `U`, `UR`, `L`, or `LR`, as in `"..."`, `R"(...)"`, `u8"..."`, `u8R"**(...)**"`, `u"..."`, `uR"*~(...)*~"`, `U"..."`, `UR"zzz(...)zzz"`, `L"..."`, or `LR"(...)"`, respectively.

2 A *string-literal* that has an `R` in the prefix is a *raw string literal*. The *d-char-sequence* serves as a delimiter. The terminating *d-char-sequence* of a *raw-string* is the same sequence of characters as the initial *d-char-sequence*. A *d-char-sequence* shall consist of at most 16 characters.

3 [*Note:* The characters `'('` and `')'` are permitted in a *raw-string*. Thus, `R"delimiter((a|b))delimiter"` is equivalent to `"(a|b)"`. — *end note*]

4 [*Note:* A source-file new-line in a raw string literal results in a new-line in the resulting execution string literal. Assuming no whitespace at the beginning of lines in the following example, the assert will succeed:

```
const char* p = R"(a\
b
c)";
assert(std::strcmp(p, "a\\\nb\nc") == 0);
```

— *end note*]

5 [*Example:* The raw string

```
R"a(
)\
a"
)a"
```

is equivalent to `"\n)\\\na\"\n"`. The raw string

```
R"(x = "\"y\"")"
```

is equivalent to `"x = \"\\\"y\\\"\""`. — *end example*]

6 After translation phase 6, a *string-literal* that does not begin with an *encoding-prefix* is an *ordinary string literal*, and is initialized with the given characters.

7 A *string-literal* that begins with `u8`, such as `u8"asdf"`, is a *UTF-8 string literal*.

8 Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals. A narrow string literal has type "array of *n* `const char`", where *n* is the size of the string as defined below, and has static storage duration (6.7).

9 For a UTF-8 string literal, each successive element of the object representation (6.9) has the value of the corresponding code unit of the UTF-8 encoding of the string.

10 A *string-literal* that begins with `u`, such as `u"asdf"`, is a `char16_t` string literal. A `char16_t` string literal has type "array of *n* `const char16_t`", where *n* is the size of the string as defined below; it is initialized with the given characters. A single *c-char* may produce more than one `char16_t` character in the form of surrogate pairs.

11 A *string-literal* that begins with `U`, such as `U"asdf"`, is a `char32_t` string literal. A `char32_t` string literal has type "array of *n* `const char32_t`", where *n* is the size of the string as defined below; it is initialized with the given characters.

12 A *string-literal* that begins with `L`, such as `L"asdf"`, is a *wide string literal*. A wide string literal has type "array of *n* `const wchar_t`", where *n* is the size of the string as defined below; it is initialized with the given characters.

13 In translation phase 6 (5.2), adjacent *string-literal*s are concatenated. If both *string-literal*s have the same *encoding-prefix*, the resulting concatenated string literal has that *encoding-prefix*. If one *string-literal* has no *encoding-prefix*, it is treated as a *string-literal* of the same *encoding-prefix* as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2017

concatenations are conditionally-supported with implementation-defined behavior. [*Note:* This concatenation is an interpretation, not a conversion. Because the interpretation happens in translation phase 6 (after each character from a string literal has been translated into a value from the appropriate character set), a *string-literal*'s initial rawness has no effect on the interpretation or well-formedness of the concatenation. — *end note*] Table 9 has some examples of valid concatenations.

Table 9 — String literal concatenations

Source		Means	Source		Means	Source		Means
`u"a"`	`u"b"`	`u"ab"`	`U"a"`	`U"b"`	`U"ab"`	`L"a"`	`L"b"`	`L"ab"`
`u"a"`	`"b"`	`u"ab"`	`U"a"`	`"b"`	`U"ab"`	`L"a"`	`"b"`	`L"ab"`
`"a"`	`u"b"`	`u"ab"`	`"a"`	`U"b"`	`U"ab"`	`"a"`	`L"b"`	`L"ab"`

Characters in concatenated strings are kept distinct.

[*Example:*

```
"\xA" "B"
```

contains the two characters `'\xA'` and `'B'` after concatenation (and not the single hexadecimal character `'\xAB'`). — *end example*]

14 After any necessary concatenation, in translation phase 7 (5.2), `'\0'` is appended to every string literal so that programs that scan a string can find its end.

15 Escape sequences and *universal-character-name*s in non-raw string literals have the same meaning as in character literals (5.13.3), except that the single quote `'` is representable either by itself or by the escape sequence `\'`, and the double quote `"` shall be preceded by a `\`, and except that a *universal-character-name* in a `char16_t` string literal may yield a surrogate pair. In a narrow string literal, a *universal-character-name* may map to more than one `char` element due to *multibyte encoding*. The size of a `char32_t` or wide string literal is the total number of escape sequences, *universal-character-name*s, and other characters, plus one for the terminating `U'\0'` or `L'\0'`. The size of a `char16_t` string literal is the total number of escape sequences, *universal-character-name*s, and other characters, plus one for each character requiring a surrogate pair, plus one for the terminating `u'\0'`. [*Note:* The size of a `char16_t` string literal is the number of code units, not the number of characters. — *end note*] Within `char32_t` and `char16_t` string literals, any *universal-character-name*s shall be within the range `0x0` to `0x10FFFF`. The size of a narrow string literal is the total number of escape sequences and other characters, plus at least one for the multibyte encoding of each *universal-character-name*, plus one for the terminating `'\0'`.

16 Evaluating a *string-literal* results in a string literal object with static storage duration, initialized from the given characters as specified above. Whether all string literals are distinct (that is, are stored in nonoverlapping objects) and whether successive evaluations of a *string-literal* yield the same or a different object is unspecified. [*Note:* The effect of attempting to modify a string literal is undefined. — *end note*]

5.13.6 Boolean literals [lex.bool]

boolean-literal:
`false`
`true`

1 The Boolean literals are the keywords `false` and `true`. Such literals are prvalues and have type `bool`.

5.13.7 Pointer literals [lex.nullptr]

pointer-literal:
`nullptr`

 ©ISO/IEC 2017 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2017