BS ISO/IEC 14882:2020

BSI Standards Publication

Programming languages — C++

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

National foreword

This British Standard is the UK implementation of ISO/IEC 14882:2020. It supersedes BS ISO/IEC 14882:2017, which is withdrawn.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 committee manager.

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 2020
Published by BSI Standards Limited 2020

ISBN 978 0 539 06199 4

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 December 2020.

Amendments/corrigenda issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

INTERNATIONAL STANDARD

ISO/IEC 14882

Sixth edition
2020-12

Programming languages — C++

Langages de programmation — C++

Reference number
ISO/IEC 14882:2020(E)

Documents
Document Sharing Hub

© ISO/IEC 2020

Full document details: BS ISO/IEC 14882:2020

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2020

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

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

© ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

Contents

Foreword x

1 Scope 1

2 Normative references 2

3 Terms and definitions 3

4 General principles 9
4.1 Implementation compliance . . . 9
4.2 Structure of this document . . . 10
4.3 Syntax notation . . . 11

5 Lexical conventions 12
5.1 Separate translation . . . 12
5.2 Phases of translation . . . 12
5.3 Character sets . . . 13
5.4 Preprocessing tokens . . . 14
5.5 Alternative tokens . . . 15
5.6 Tokens . . . 15
5.7 Comments . . . 15
5.8 Header names . . . 16
5.9 Preprocessing numbers . . . 16
5.10 Identifiers . . . 16
5.11 Keywords . . . 17
5.12 Operators and punctuators . . . 18
5.13 Literals . . . 19

6 Basics 28
6.1 Preamble . . . 28
6.2 Declarations and definitions . . . 28
6.3 One-definition rule . . . 30
6.4 Scope . . . 34
6.5 Name lookup . . . 40
6.6 Program and linkage . . . 53
6.7 Memory and objects . . . 57
6.8 Types . . . 71
6.9 Program execution . . . 78

7 Expressions 90
7.1 Preamble . . . 90
7.2 Properties of expressions . . . 91
7.3 Standard conversions . . . 94
7.4 Usual arithmetic conversions . . . 99
7.5 Primary expressions . . . 99
7.6 Compound expressions . . . 116
7.7 Constant expressions . . . 147

8 Statements 153
8.1 Preamble . . . 153
8.2 Labeled statement . . . 154
8.3 Expression statement . . . 154
8.4 Compound statement or block . . . 154
8.5 Selection statements . . . 154

 ©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

8.6 Iteration statements . . . 156
8.7 Jump statements . . . 159
8.8 Declaration statement . . . 161
8.9 Ambiguity resolution . . . 161

9 Declarations **163**
9.1 Preamble . . . 163
9.2 Specifiers . . . 165
9.3 Declarators . . . 182
9.4 Initializers . . . 197
9.5 Function definitions . . . 213
9.6 Structured binding declarations . . . 219
9.7 Enumerations . . . 220
9.8 Namespaces . . . 224
9.9 The `using` declaration . . . 231
9.10 The `asm` declaration . . . 236
9.11 Linkage specifications . . . 237
9.12 Attributes . . . 239

10 Modules **247**
10.1 Module units and purviews . . . 247
10.2 Export declaration . . . 248
10.3 Import declaration . . . 251
10.4 Global module fragment . . . 252
10.5 Private module fragment . . . 254
10.6 Instantiation context . . . 255
10.7 Reachability . . . 256

11 Classes **258**
11.1 Preamble . . . 258
11.2 Properties of classes . . . 259
11.3 Class names . . . 260
11.4 Class members . . . 262
11.5 Unions . . . 284
11.6 Local class declarations . . . 287
11.7 Derived classes . . . 287
11.8 Member name lookup . . . 295
11.9 Member access control . . . 298
11.10 Initialization . . . 308
11.11 Comparisons . . . 319
11.12 Free store . . . 322

12 Overloading **324**
12.1 Preamble . . . 324
12.2 Overloadable declarations . . . 324
12.3 Declaration matching . . . 326
12.4 Overload resolution . . . 327
12.5 Address of overloaded function . . . 351
12.6 Overloaded operators . . . 352
12.7 Built-in operators . . . 356
12.8 User-defined literals . . . 358

13 Templates **360**
13.1 Preamble . . . 360
13.2 Template parameters . . . 361
13.3 Names of template specializations . . . 365
13.4 Template arguments . . . 368
13.5 Template constraints . . . 373
13.6 Type equivalence . . . 379

©ISO/IEC 2020 – All rights reserved
Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

13.7 Template declarations . . . 380
13.8 Name resolution . . . 401
13.9 Template instantiation and specialization . . . 417
13.10 Function template specializations . . . 431

14 Exception handling **451**
14.1 Preamble . . . 451
14.2 Throwing an exception . . . 452
14.3 Constructors and destructors . . . 453
14.4 Handling an exception . . . 454
14.5 Exception specifications . . . 456
14.6 Special functions . . . 458

15 Preprocessing directives **460**
15.1 Preamble . . . 460
15.2 Conditional inclusion . . . 462
15.3 Source file inclusion . . . 464
15.4 Module directive . . . 465
15.5 Header unit importation . . . 466
15.6 Macro replacement . . . 467
15.7 Line control . . . 472
15.8 Error directive . . . 473
15.9 Pragma directive . . . 473
15.10 Null directive . . . 473
15.11 Predefined macro names . . . 473
15.12 Pragma operator . . . 475

16 Library introduction **477**
16.1 General . . . 477
16.2 The C standard library . . . 478
16.3 Method of description . . . 478
16.4 Library-wide requirements . . . 484

17 Language support library **505**
17.1 General . . . 505
17.2 Common definitions . . . 505
17.3 Implementation properties . . . 509
17.4 Integer types . . . 519
17.5 Startup and termination . . . 520
17.6 Dynamic memory management . . . 522
17.7 Type identification . . . 529
17.8 Source location . . . 530
17.9 Exception handling . . . 532
17.10 Initializer lists . . . 536
17.11 Comparisons . . . 537
17.12 Coroutines . . . 545
17.13 Other runtime support . . . 549

18 Concepts library **552**
18.1 General . . . 552
18.2 Equality preservation . . . 552
18.3 Header `<concepts>` synopsis . . . 553
18.4 Language-related concepts . . . 555
18.5 Comparison concepts . . . 560
18.6 Object concepts . . . 563
18.7 Callable concepts . . . 563

 ©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

19 Diagnostics library **565**
19.1 General . . . 565
19.2 Exception classes . . . 565
19.3 Assertions . . . 568
19.4 Error numbers . . . 568
19.5 System error support . . . 570

20 General utilities library **579**
20.1 General . . . 579
20.2 Utility components . . . 579
20.3 Compile-time integer sequences . . . 584
20.4 Pairs . . . 585
20.5 Tuples . . . 589
20.6 Optional objects . . . 599
20.7 Variants . . . 611
20.8 Storage for any type . . . 622
20.9 Bitsets . . . 627
20.10 Memory . . . 632
20.11 Smart pointers . . . 648
20.12 Memory resources . . . 671
20.13 Class template `scoped_allocator_adaptor` . . . 680
20.14 Function objects . . . 684
20.15 Metaprogramming and type traits . . . 707
20.16 Compile-time rational arithmetic . . . 732
20.17 Class `type_index` . . . 734
20.18 Execution policies . . . 736
20.19 Primitive numeric conversions . . . 738
20.20 Formatting . . . 740

21 Strings library **759**
21.1 General . . . 759
21.2 Character traits . . . 759
21.3 String classes . . . 764
21.4 String view classes . . . 790
21.5 Null-terminated sequence utilities . . . 800

22 Containers library **805**
22.1 General . . . 805
22.2 Container requirements . . . 805
22.3 Sequence containers . . . 839
22.4 Associative containers . . . 867
22.5 Unordered associative containers . . . 885
22.6 Container adaptors . . . 906
22.7 Views . . . 914

23 Iterators library **921**
23.1 General . . . 921
23.2 Header `<iterator>` synopsis . . . 921
23.3 Iterator requirements . . . 928
23.4 Iterator primitives . . . 948
23.5 Iterator adaptors . . . 951
23.6 Stream iterators . . . 972
23.7 Range access . . . 978

24 Ranges library **980**
24.1 General . . . 980
24.2 Header `<ranges>` synopsis . . . 980
24.3 Range access . . . 985
24.4 Range requirements . . . 989

 ©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

24.5 Range utilities . . . 992
24.6 Range factories . . . 997
24.7 Range adaptors . . . 1007

25 Algorithms library **1044**
25.1 General . . . 1044
25.2 Algorithms requirements . . . 1044
25.3 Parallel algorithms . . . 1046
25.4 Header `<algorithm>` synopsis . . . 1049
25.5 Algorithm result types . . . 1084
25.6 Non-modifying sequence operations . . . 1087
25.7 Mutating sequence operations . . . 1099
25.8 Sorting and related operations . . . 1115
25.9 Header `<numeric>` synopsis . . . 1142
25.10 Generalized numeric operations . . . 1145
25.11 Specialized `<memory>` algorithms . . . 1154
25.12 C library algorithms . . . 1160

26 Numerics library **1161**
26.1 General . . . 1161
26.2 Numeric type requirements . . . 1161
26.3 The floating-point environment . . . 1161
26.4 Complex numbers . . . 1162
26.5 Bit manipulation . . . 1170
26.6 Random number generation . . . 1173
26.7 Numeric arrays . . . 1210
26.8 Mathematical functions for floating-point types . . . 1229
26.9 Numbers . . . 1244

27 Time library **1245**
27.1 General . . . 1245
27.2 Header `<chrono>` synopsis . . . 1245
27.3 *Cpp17Clock* requirements . . . 1259
27.4 Time-related traits . . . 1259
27.5 Class template `duration` . . . 1261
27.6 Class template `time_point` . . . 1268
27.7 Clocks . . . 1271
27.8 The civil calendar . . . 1282
27.9 Class template `hh_mm_ss` . . . 1311
27.10 12/24 hours functions . . . 1313
27.11 Time zones . . . 1313
27.12 Formatting . . . 1326
27.13 Parsing . . . 1330
27.14 Header `<ctime>` synopsis . . . 1334

28 Localization library **1335**
28.1 General . . . 1335
28.2 Header `<locale>` synopsis . . . 1335
28.3 Locales . . . 1336
28.4 Standard `locale` categories . . . 1342
28.5 C library locales . . . 1374

29 Input/output library **1375**
29.1 General . . . 1375
29.2 Iostreams requirements . . . 1375
29.3 Forward declarations . . . 1376
29.4 Standard iostream objects . . . 1378
29.5 Iostreams base classes . . . 1380
29.6 Stream buffers . . . 1395

 ©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

29.7 Formatting and manipulators . . . 1403
29.8 String-based streams . . . 1427
29.9 File-based streams . . . 1441
29.10 Synchronized output streams . . . 1453
29.11 File systems . . . 1458
29.12 C library files . . . 1503

30 Regular expressions library **1506**
30.1 General . . . 1506
30.2 Definitions . . . 1506
30.3 Requirements . . . 1507
30.4 Header `<regex>` synopsis . . . 1508
30.5 Namespace `std::regex_constants` . . . 1512
30.6 Class `regex_error` . . . 1515
30.7 Class template `regex_traits` . . . 1515
30.8 Class template `basic_regex` . . . 1517
30.9 Class template `sub_match` . . . 1522
30.10 Class template `match_results` . . . 1523
30.11 Regular expression algorithms . . . 1528
30.12 Regular expression iterators . . . 1532
30.13 Modified ECMAScript regular expression grammar . . . 1537

31 Atomic operations library **1540**
31.1 General . . . 1540
31.2 Header `<atomic>` synopsis . . . 1540
31.3 Type aliases . . . 1544
31.4 Order and consistency . . . 1544
31.5 Lock-free property . . . 1546
31.6 Waiting and notifying . . . 1546
31.7 Class template `atomic_ref` . . . 1547
31.8 Class template `atomic` . . . 1553
31.9 Non-member functions . . . 1568
31.10 Flag type and operations . . . 1568
31.11 Fences . . . 1570

32 Thread support library **1572**
32.1 General . . . 1572
32.2 Requirements . . . 1572
32.3 Stop tokens . . . 1574
32.4 Threads . . . 1579
32.5 Mutual exclusion . . . 1586
32.6 Condition variables . . . 1605
32.7 Semaphore . . . 1612
32.8 Coordination types . . . 1614
32.9 Futures . . . 1617

Annex A Grammar summary **1632**
A.1 General . . . 1632
A.2 Keywords . . . 1632
A.3 Lexical conventions . . . 1632
A.4 Basics . . . 1636
A.5 Expressions . . . 1637
A.6 Statements . . . 1641
A.7 Declarations . . . 1641
A.8 Modules . . . 1647
A.9 Classes . . . 1648
A.10 Overloading . . . 1649
A.11 Templates . . . 1649
A.12 Exception handling . . . 1651

 ©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

A.13 Preprocessing directives . . . 1651

Annex B Implementation quantities **1653**

Annex C Compatibility **1655**
C.1 C++ and ISO C++ 2017 . . . 1655
C.2 C++ and ISO C++ 2014 . . . 1662
C.3 C++ and ISO C++ 2011 . . . 1666
C.4 C++ and ISO C++ 2003 . . . 1667
C.5 C++ and ISO C . . . 1673
C.6 C standard library . . . 1681

Annex D Compatibility features **1683**
D.1 General . . . 1683
D.2 Arithmetic conversion on enumerations . . . 1683
D.3 Implicit capture of `*this` by reference . . . 1683
D.4 Comma operator in subscript expressions . . . 1683
D.5 Array comparisons . . . 1683
D.6 Deprecated `volatile` types . . . 1684
D.7 Redeclaration of `static constexpr` data members . . . 1684
D.8 Non-local use of TU-local entities . . . 1685
D.9 Implicit declaration of copy functions . . . 1685
D.10 C headers . . . 1685
D.11 Requires paragraph . . . 1686
D.12 Relational operators . . . 1686
D.13 `char*` streams . . . 1687
D.14 Deprecated type traits . . . 1694
D.15 Tuple . . . 1695
D.16 Variant . . . 1695
D.17 Deprecated `iterator` class template . . . 1696
D.18 Deprecated `move_iterator` access . . . 1696
D.19 Deprecated `shared_ptr` atomic access . . . 1696
D.20 Deprecated `basic_string` capacity . . . 1698
D.21 Deprecated standard code conversion facets . . . 1698
D.22 Deprecated convenience conversion interfaces . . . 1700
D.23 Deprecated locale category facets . . . 1703
D.24 Deprecated filesystem path factory functions . . . 1704
D.25 Deprecated atomic operations . . . 1704

Bibliography **1706**

Cross references **1707**

Cross references from ISO C++ 2017 **1731**

Index **1734**

Index of grammar productions **1768**

Index of library headers **1773**

Index of library names **1775**

Index of library concepts **1847**

Index of implementation-defined behavior **1850**

 ©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

Foreword

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

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

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC 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) or the IEC list of patent declarations received (see http://patents.iec.ch).

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 www.iso.org/iso/foreword.html.

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

This sixth edition cancels and replaces the fifth edition (ISO/IEC 14882:2017), which has been technically revised.

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

— inclusion of the provisions of ISO/IEC TS 19217:2015, ISO/IEC TS 21425:2017, ISO/IEC TS 22277:2017, ISO/IEC TS 21544:2018, portions of ISO/IEC TS 19571:2016, and portions of ISO/IEC TS 19568:2017.

— addition of concepts, *requires-clauses*, *requires-expressions*, and `<concepts>` (18.3) header

— addition of coroutines, including `co_yield`, `co_await`, and `co_return` keywords and `<coroutine>` (17.12.2) header

— addition of modules, *import-declarations*, and *export-declarations*

— addition of three-way comparison, defaulted comparisons, rewriting of comparison operator expressions, and `<compare>` (17.11.1) header

— addition of designated initializers

— support for class types and floating-point types as the type of a non-type template parameter

— new attributes `[[no_unique_address]]`, `[[likely]]`, `[[unlikely]]`

— support for optional reason string in `[[nodiscard]]` attribute

— ability to require constant initialization with `constinit` keyword

— ability to require constant evaluation with `consteval` keyword

— extensions to constant evaluation

— support for controlling destruction in a class-specific operator delete function

— addition of `using enum` declaration

— addition of `char8_t` type

— support for an initializer statement in range-based for loops

 ©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

— support for default member initializers for bit-fields
— support for parenthesized aggregate initialization
— extensions to lambda expressions
— extensions to structured bindings
— support for inline namespaces in nested namespace definitions
— support for conditionally-explicit member functions
— extensions to class template argument deduction
— reduced cases in which `typename` is required
— support for calling an undeclared *template-id* via argument-dependent name lookup
— revised memory model
— extended support for variadic macros with `__VA_OPT__`
— feature test macros and `<version>` (17.3.2) header
— addition of ranges and `<ranges>` (24.2) header
— addition of calendar and time zone support
— addition of text formatting library and `<format>` (20.20.1) header
— addition of `<barrier>` (32.8.3.2), `<latch>` (32.8.2.2), and `<semaphore>` (32.7.2) headers
— addition of mathematical constants library and `<numbers>` (26.9.1) header
— support for representing source locations and `<source_location>` (17.8.1) header
— addition of `span` view and `<span>` (22.7.2) header
— addition of joining thread class and `<stop_token>` (32.3.2) header
— extensions to atomic types and operations
— addition of `unsequenced` execution policy
— new utility functions, types, and templates in the standard library
— addition of bit manipulation library and `<bit>` (26.5.2) header
— addition of a synchronized buffered output stream and `<syncstream>` (29.10.1) header
— support for heterogeneous lookup for unordered containers
— support for element existence detection in associative containers
— support for move semantics in `<numeric>` (25.9) algorithms
— support for efficient access to the buffer of a `basic_stringbuf`
— extended constant expression evaluation support in the standard library

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

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

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 9899:2018 *Programming languages — C* (hereinafter referred to as the *C standard*). C++ provides many facilities beyond those provided by C, including additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

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) — ISO/IEC 2382, *Information technology — Vocabulary*

(1.2) — ISO 8601:2004, *Data elements and interchange formats — Information interchange — Representation of dates and times*

(1.3) — ISO/IEC 9899:2018, *Programming languages — C*

(1.4) — ISO/IEC 9945:2003, *Information Technology — Portable Operating System Interface (POSIX*[1]*)*

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

(1.6) — ISO/IEC 10646:2003,[2] *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*

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

(1.8) — Ecma International, *ECMAScript*[3] *Language Specification*, Standard Ecma-262, third edition, 1999.

2 The library described in ISO/IEC 9899:2018, Clause 7, is hereinafter called the *C standard library*.[4]

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*.

5 [*Note 1*: References to ISO/IEC 10646:2003 are used only to support deprecated features (D.21). — *end note*]

1) POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

2) Cancelled and replaced by ISO/IEC 10646:2017.

3) ECMAScript® is a registered trademark of Ecma International. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO or IEC of this product.

4) With the qualifications noted in Clause 17 through Clause 32 and in C.6, the C standard library is a subset of the C++ standard library.

 ©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

3 Terms and definitions [intro.defs]

1 For the purposes of this document, the terms and definitions given in ISO/IEC 2382, 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 https://www.iso.org/obp

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

3 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

[*Note 1 to entry*: Only objects of scalar type can be accessed. Reads of scalar objects are described in 7.3.2 and modifications of scalar objects are described in 7.6.19, 7.6.1.6, and 7.6.2.3. Attempts to read or modify an object of class type typically invoke a constructor (11.4.5) or assignment operator (11.4.6); such invocations do not themselves constitute accesses, although they may involve accesses of scalar subobjects. — *end note*]

3.2 **[defns.arbitrary.stream]**

arbitrary-positional stream

⟨library⟩ stream that can seek to any integral position within the length of the stream

[*Note 1 to entry*: Every arbitrary-positional stream is also a repositional stream (3.42). — *end note*]

3.3 **[defns.argument]**

argument

⟨function call expression⟩ expression in the comma-separated list bounded by the parentheses

3.4 **[defns.argument.macro]**

argument

⟨function-like macro⟩ sequence of preprocessing tokens in the comma-separated list bounded by the parentheses

3.5 **[defns.argument.throw]**

argument

⟨throw expression⟩ operand of `throw`

3.6 **[defns.argument.templ]**

argument

⟨template instantiation⟩ *constant-expression*, *type-id*, or *id-expression* in the comma-separated list bounded by the angle brackets

3.7 **[defns.block]**

block

⟨execution⟩ 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

3.8 **[defns.block.stmt]**

block

⟨statement⟩ compound statement

3.9 **[defns.character]**

character

⟨library⟩ object which, when treated sequentially, can represent text

[*Note 1 to entry*: The term does not mean only `char`, `char8_t`, `char16_t`, `char32_t`, and `wchar_t` objects (6.8.2), but any value that can be represented by a type that provides the definitions specified in Clause 21, Clause 28, Clause 29, or Clause 30. — *end note*]

 ©ISO/IEC 2020 – All rights reserved

AVA Documents Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

3.10 **[defns.character.container]**

character container type

⟨library⟩ class or a type used to represent a character

[*Note 1 to entry*: It is used for one of the template parameters of the string, iostream, and regular expression class templates. — *end note*]

3.11 **[defns.component]**

component

⟨library⟩ group of library entities directly related as members, parameters, or return types

[*Note 1 to entry*: For example, the class template `basic_string` and the non-member function templates that operate on strings are referred to as the *string component*. — *end note*]

3.12 **[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*]

3.13 **[defns.const.subexpr]**

constant subexpression

expression whose evaluation as subexpression of a *conditional-expression* `CE` would not prevent `CE` from being a core constant expression

3.14 **[defns.deadlock]**

deadlock

⟨library⟩ situation wherein one or more threads are unable to continue execution because each is blocked waiting for one or more of the others to satisfy some condition

3.15 **[defns.default.behavior.impl]**

default behavior

⟨library implementation⟩ specific behavior provided by the implementation, within the scope of the required behavior

3.16 **[defns.diagnostic]**

diagnostic message

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

3.17 **[defns.direct-non-list-init]**

direct-non-list-initialization

direct-initialization that is not list-initialization

3.18 **[defns.dynamic.type]**

dynamic type

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

[*Example 1*: If a pointer (9.3.4.2) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (11.7), the dynamic type of the expression `*p` is "`D`". References (9.3.4.3) are treated similarly. — *end example*]

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

dynamic type

⟨prvalue⟩ static type of the prvalue expression

3.20 **[defns.expression-equivalent]**

expression-equivalent

⟨library⟩ expressions that all have the same effects, either are all potentially-throwing or are all not potentially-throwing, and either are all constant subexpressions or are all not constant subexpressions

[*Example 1*: For a value `x` of type `int` and a function `f` that accepts integer arguments, the expressions `f(x + 2)`, `f(2 + x)`, and `f(1 + x + 1)` are expression-equivalent. — *end example*]

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

3.21 **[defns.handler]**
handler function
⟨library⟩ non-reserved function whose definition may be provided by a C++ program

[*Note 1 to entry*: A C++ program may designate a handler function at various points in its execution by supplying a pointer to the function when calling any of the library functions that install handler functions (Clause 17). — *end note*]

3.22 **[defns.ill.formed]**
ill-formed program
program that is not well-formed (3.60)

3.23 **[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

3.24 **[defns.order.ptr]**
implementation-defined strict total order over pointers
⟨library⟩ implementation-defined strict total ordering over all pointer values such that the ordering is consistent with the partial order imposed by the builtin operators `<`, `>`, `<=`, `>=`, and `<=>`

3.25 **[defns.impl.limits]**
implementation limits
restrictions imposed upon programs by the implementation

3.26 **[defns.iostream.templates]**
iostream class templates
⟨library⟩ templates that are declared in header `<iosfwd>` and take two template arguments

[*Note 1 to entry*: The arguments are named `charT` and `traits`. The argument `charT` is a character container class, and the argument `traits` is a class which defines additional characteristics and functions of the character type represented by `charT` necessary to implement the iostream class templates. — *end note*]

3.27 **[defns.locale.specific]**
locale-specific behavior
behavior that depends on local conventions of nationality, culture, and language that each implementation documents

3.28 **[defns.modifier]**
modifier function
⟨library⟩ class member function other than a constructor, assignment operator, or destructor that alters the state of an object of the class

3.29 **[defns.move.assign]**
move assignment
⟨library⟩ assignment of an rvalue of some object type to a modifiable lvalue of the same type

3.30 **[defns.move.constr]**
move construction
⟨library⟩ direct-initialization of an object of some type with an rvalue of the same type

3.31 **[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*]

3.32 **[defns.ntcts]**
NTCTS
⟨library⟩ sequence of values that have character type that precede the terminating null character type value `charT()`

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

3.33 **[defns.observer]**

observer function

⟨library⟩ class member function that accesses the state of an object of the class but does not alter that state

[*Note 1 to entry*: Observer functions are specified as `const` member functions (11.4.3.2). — *end note*]

3.34 **[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

3.35 **[defns.parameter.macro]**

parameter

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

3.36 **[defns.parameter.templ]**

parameter

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

3.37 **[defns.prog.def.spec]**

program-defined specialization

⟨library⟩ explicit template specialization or partial specialization that is not part of the C++ standard library and not defined by the implementation

3.38 **[defns.prog.def.type]**

program-defined type

⟨library⟩ non-closure class type or enumeration type that is not part of the C++ standard library and not defined by the implementation, or a closure type of a non-implementation-provided lambda expression, or an instantiation of a program-defined specialization

[*Note 1 to entry*: Types defined by the implementation include extensions (4.1) and internal types used by the library. — *end note*]

3.39 **[defns.projection]**

projection

⟨library⟩ transformation that an algorithm applies before inspecting the values of elements

[*Example 1*:

```
std::pair<int, std::string_view> pairs[] = {{2, "foo"}, {1, "bar"}, {0, "baz"}};
std::ranges::sort(pairs, std::ranges::less{}, [](auto const& p) { return p.first; });
```

sorts the pairs in increasing order of their `first` members:

```
{{0, "baz"}, {1, "bar"}, {2, "foo"}}
```

— *end example*]

3.40 **[defns.referenceable]**

referenceable type

type that is either an object type, a function type that does not have cv-qualifiers or a *ref-qualifier*, or a reference type

[*Note 1 to entry*: The term describes a type to which a reference can be created, including reference types. — *end note*]

3.41 **[defns.replacement]**

replacement function

⟨library⟩ non-reserved function whose definition is provided by a C++ program

[*Note 1 to entry*: Only one definition for such a function is in effect for the duration of the program's execution, as the result of creating the program (5.2) and resolving the definitions of all translation units (6.6). — *end note*]

3.42 **[defns.repositional.stream]**

repositional stream

⟨library⟩ stream that can seek to a position that was previously encountered

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

3.43 **[defns.required.behavior]**

required behavior

⟨library⟩ description of replacement function and handler function semantics applicable to both the behavior provided by the implementation and the behavior of any such function definition in the program

[*Note 1 to entry*: If such a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined. — *end note*]

3.44 **[defns.reserved.function]**

reserved function

⟨library⟩ function, specified as part of the C++ standard library, that is defined by the implementation

[*Note 1 to entry*: If a C++ program provides a definition for any reserved function, the results are undefined. — *end note*]

3.45 **[defns.signature]**

signature

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

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

3.46 **[defns.signature.friend]**

signature

⟨non-template friend function with trailing *requires-clause*⟩ name, parameter-type-list, enclosing class, and trailing *requires-clause*

3.47 **[defns.signature.templ]**

signature

⟨function template⟩ name, parameter-type-list, enclosing namespace (if any), return type, *template-head*, and trailing *requires-clause* (if any)

3.48 **[defns.signature.templ.friend]**

signature

⟨friend function template with constraint involving enclosing template parameters⟩ name, parameter-type-list, return type, enclosing class, *template-head*, and trailing *requires-clause* (if any)

3.49 **[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)

3.50 **[defns.signature.member]**

signature

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

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

signature

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

3.52 **[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)

3.53 **[defns.stable]**

stable algorithm

⟨library⟩ algorithm that preserves, as appropriate to the particular algorithm, the order of elements

[*Note 1 to entry*: Requirements for stable algorithms are given in 16.4.6.8. — *end note*]

 ©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

3.54 **[defns.static.type]**

static type

type of an expression 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*]

3.55 **[defns.traits]**

traits class

⟨library⟩ class that encapsulates a set of types and functions necessary for class templates and function templates to manipulate objects of types for which they are instantiated

3.56 **[defns.unblock]**

unblock

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

3.57 **[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 (7.7) never exhibits behavior explicitly specified as undefined in Clause 4 through Clause 15. — *end note*]

3.58 **[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*]

3.59 **[defns.valid]**

valid but unspecified state

⟨library⟩ value of an object that is not specified except that the object's invariants are met and operations on the object behave as specified for its type

[*Example 1*: If an object `x` of type `std::vector<int>` is in a valid but unspecified state, `x.empty()` can be called unconditionally, and `x.front()` can be called only if `x.empty()` returns `false`. — *end example*]

3.60 **[defns.well.formed]**

well-formed program

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

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

4 General principles [intro]

4.1 Implementation compliance [intro.compliance]

4.1.1 General [intro.compliance.general]

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 Clause 5 through Clause 32 and Annex D, a conforming implementation shall, within its resource limits as described in Annex B, accept and correctly execute[5] 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 1*: During template argument deduction and substitution, certain constructs that in other contexts require a diagnostic are treated differently; see 13.10.3. — *end note*]

3 For classes and class templates, the library Clauses specify partial definitions. Private members (11.9) 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 (9.8). A C++ translation unit (5.2) obtains access to these names by including the appropriate standard library header or importing the appropriate standard library named header unit (16.4.3.2).

6 The templates, classes, functions, and objects in the library have external linkage (6.6). 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 (16.4.2.4).

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.[6]

5) "Correct execution" can include undefined behavior, depending on the data being processed; see Clause 3 and 6.9.1.
6) This documentation also defines implementation-defined behavior; see 4.1.2.

 ©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

4.1.2 Abstract machine [intro.abstract]

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.[7]

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.[8] 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, order of evaluation of arguments in a function call (7.6.1.3)). 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 1*: 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 The least requirements on a conforming implementation are:

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

(6.2) — 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.

(6.3) — 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 2*: More stringent correspondences between abstract and actual semantics can be defined by each implementation. — *end note*]

4.2 Structure of this document [intro.structure]

1 Clause 5 through Clause 15 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 Clause 17 through Clause 32 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 16.

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.

7) 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.

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

 ©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

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).

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

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 (16.4.2.3) and source files included (15.3) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (15.2) preprocessing directives, is called a *translation unit*.

[*Note 1*: A C++ program need not all be translated at the same time. — *end note*]

2 [*Note 2*: Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (6.6) by (for example) calls to functions whose identifiers have external or module linkage, manipulation of objects whose identifiers have external or module linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program (6.6). — *end note*]

5.2 Phases of translation [lex.phases]

1 The precedence among the syntax rules of translation is specified by the following phases.[9]

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.[10] 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 1*: 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 (15.6.4), the behavior is undefined. A `#include` preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

5. Each basic 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.[11]

9) Implementations behave as if these separate phases occur, although in practice different phases can be folded together.

10) 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.

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

 ©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

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 1*: The process of analyzing and translating the tokens can occasionally result in one token being replaced by a sequence of other tokens (13.3). — *end note*]

   It is implementation-defined whether the sources for module units and header units on which the current translation unit has an interface dependency (10.1, 10.3) are required to be available.

   [*Note 2*: 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 3*: Some or all of these can be supplied from a library. — *end note*]

   Each translated translation unit is examined to produce a list of required instantiations.

   [*Note 4*: This can include instantiations which have been explicitly requested (13.9.3). — *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 5*: An implementation can choose to 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 6*: 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:[12]

```
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*

A *universal-character-name* designates the character in ISO/IEC 10646 (if any) whose code point is the hexadecimal number represented by the sequence of *hexadecimal-digit*s in the *universal-character-name*. The program is ill-formed if that number is not a code point or if it is a surrogate code point. Noncharacter code points and reserved code points are considered to designate separate characters distinct from any ISO/IEC 10646 character. If a *universal-character-name* outside the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of a *character-literal* or *string-literal* (in either case, including within a *user-defined-literal*) corresponds to a control character or to a character in the basic source character set, the program is ill-formed.[13]

12) 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, the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, and therefore implementations must document how the basic source characters are represented in source files.

13) 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 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

[*Note 1*: ISO/IEC 10646 code points are integers in the range [0, 10FFFF] (hexadecimal). A surrogate code point is a value in the range [D800, DFFF] (hexadecimal). A control character is a character whose code point is in either of the ranges [0, 1F] or [7F, 9F] (hexadecimal). — *end note*]

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*
- *import-keyword*
- *module-keyword*
- *export-keyword*
- *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, or an operator or 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, placeholder tokens produced by preprocessing `import` and `module` directives (*import-keyword*, *module-keyword*, and *export-keyword*), 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 15, 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 can 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

$$\textit{encoding-prefix}_{opt}\ \texttt{R}\ \textit{raw-string}$$

(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 matches the syntax of a preprocessing token, even if that would cause further lexical analysis to fail, except that a *header-name* (5.8) is only formed

(3.3.1) — after the `include` or `import` preprocessing token in an `#include` (15.3) or `import` (15.5) directive, or

(3.3.2) — within a *has-include-expression*.

©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

[*Example 1*:

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

— *end example*]

4 The *import-keyword* is produced by processing an `import` directive (15.5), the *module-keyword* is produced by preprocessing a `module` directive (15.4), and the *export-keyword* is produced by preprocessing either of the previous two directives.

[*Note 1*: None has any observable spelling. — *end note*]

5 [*Example 2*: The program fragment `0xe+foo` is parsed as a preprocessing number token (one that is not a valid *integer-literal* or *floating-point-literal* token), even though a parse as three preprocessing tokens `0xe`, `+`, and `foo` can produce a valid expression (for example, if `foo` is a macro defined as `1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid *floating-point-literal* token), whether or not `E` is a macro name. — *end example*]

6 [*Example 3*: 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` can yield a correct expression. — *end example*]

5.5 Alternative tokens [lex.digraph]

1 Alternative token representations are provided for some operators and punctuators.[14]

2 In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.[15] The set of alternative tokens is defined in Table 1.

Table 1: Alternative tokens [tab:lex.digraph]

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-or-punctuator*

1 There are five kinds of tokens: identifiers, keywords, literals,[16] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "white space"), as described below, are ignored except as they serve to separate tokens.

[*Note 1*: 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.

14) 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".

15) Thus the "stringized" values (15.6.3) of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

16) Literals include strings and character and numeric literals.

©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

[*Note 1*: 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 1*: Header name preprocessing tokens only appear within a `#include` preprocessing directive, a `__has_include` preprocessing expression, or after certain occurrences of an `import` token (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 15.3.

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*.[17]

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 .

1 Preprocessing number tokens lexically include all *integer-literal* tokens (5.13.2) and all *floating-point-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-point-literal* token.

5.10 Identifiers [lex.name]

identifier:
  identifier-nondigit
  identifier identifier-nondigit
  identifier digit

identifier-nondigit:
  nondigit
  universal-character-name

17) Thus, a sequence of characters that resembles an escape sequence can 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 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

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/IEC 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.[18]

Table 2: Ranges of characters allowed [tab:lex.name.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) [tab:lex.name.disallowed]

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

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 [tab:lex.name.special]

`final`	`import`	`module`	`override`

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]

keyword:
- any identifier listed in Table 5
- *import-keyword*
- *module-keyword*
- *export-keyword*

18) On systems in which linkers cannot accept extended characters, an encoding of the *universal-character-name* can be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters can be used to encode the `\u` in a *universal-character-name*. Extended characters can 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 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

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* (9.12.1).

[*Note 1*: The `register` keyword is unused but is reserved for future use. — *end note*]

Table 5: Keywords [tab:lex.key]

alignas	constinit	false	public	true
alignof	const_cast	float	register	try
asm	continue	for	reinterpret_cast	typedef
auto	co_await	friend	requires	typeid
bool	co_return	goto	return	typename
break	co_yield	if	short	union
case	decltype	inline	signed	unsigned
catch	default	int	sizeof	using
char	delete	long	static	virtual
char8_t	do	mutable	static_assert	void
char16_t	double	namespace	static_cast	volatile
char32_t	dynamic_cast	new	struct	wchar_t
class	else	noexcept	switch	while
concept	enum	nullptr	template	
const	explicit	operator	this	
consteval	export	private	thread_local	
constexpr	extern	protected	throw	

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 [tab:lex.key.digraph]

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	xor	xor_eq	

5.12 Operators and punctuators [lex.operators]

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

preprocessing-op-or-punc:
  preprocessing-operator
  operator-or-punctuator

preprocessing-operator: one of
```
#      ##     %:     %:%:
```

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

Each *operator-or-punctuator* is converted to a single token in translation phase 7 (5.2).

©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

5.13 Literals [lex.literal]

5.13.1 Kinds of literals [lex.literal.kinds]

1 There are several kinds of literals.[19]

literal:
- *integer-literal*
- *character-literal*
- *floating-point-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: one of
`0 1`

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`

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

©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

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

1 In an *integer-literal*, the sequence of *binary-digit*s, *octal-digit*s, *digit*s, or *hexadecimal-digit*s is interpreted as a base N integer as shown in table Table 7; the lexically first digit of the sequence of digits is the most significant.

[*Note 1*: The prefix and any optional separating single quotes are ignored when determining the value. — *end note*]

Table 7: Base of *integer-literal*s [tab:lex.icon.base]

Kind of *integer-literal*	base N
binary-literal	2
octal-literal	8
decimal-literal	10
hexadecimal-literal	16

2 The *hexadecimal-digit*s `a` through `f` and `A` through `F` have decimal values ten through fifteen.

[*Example 1*: The number twelve can be written `12`, `014`, `0XC`, or `0b1100`. The *integer-literal*s `1048576`, `1'048'576`, `0X100000`, `0x10'0000`, and `0'004'000'000` all have the same value. — *end example*]

3 The type of an *integer-literal* is the first type in the list in Table 8 corresponding to its optional *integer-suffix* in which its value can be represented. An *integer-literal* is a prvalue.

Table 8: Types of *integer-literal*s [tab:lex.icon.type]

integer-suffix	*decimal-literal*	*integer-literal* other than *decimal-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`

4 If an *integer-literal* cannot be represented by any type in its list and an extended integer type (6.8.2) 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.

5.13.3 Character literals [lex.ccon]

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

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

©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

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

c-char:
  any member of the basic 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

1 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.

2 A *character-literal* that begins with `u8`, such as `u8'w'`, is a *character-literal* of type `char8_t`, known as a *UTF-8 character literal*. The value of a UTF-8 character literal is equal to its ISO/IEC 10646 code point value, provided that the code point value can be encoded as a single UTF-8 code unit.

[*Note 1*: That is, provided the code point value is in the range [0, 7F] (hexadecimal). — *end note*]

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.

3 A *character-literal* that begins with the letter `u`, such as `u'x'`, is a *character-literal* of type `char16_t`, known as a *UTF-16 character literal*. The value of a UTF-16 character literal is equal to its ISO/IEC 10646 code point value, provided that the code point value is representable with a single 16-bit code unit.

[*Note 2*: That is, provided the code point value is in the range [0, FFFF] (hexadecimal). — *end note*]

If the value is not representable with a single 16-bit code unit, the program is ill-formed. A UTF-16 character literal containing multiple *c-char*s is ill-formed.

4 A *character-literal* that begins with the letter `U`, such as `U'y'`, is a *character-literal* of type `char32_t`, known as a *UTF-32 character literal*. The value of a UTF-32 character literal containing a single *c-char* is equal to its ISO/IEC 10646 code point value. A UTF-32 character literal containing multiple *c-char*s is ill-formed.

5 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`.[20] 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 3*: The type `wchar_t` is able to represent all members of the execution wide-character set (see 6.8.2). — *end note*]

The value of a wide-character literal containing multiple *c-char*s is implementation-defined.

6 Certain non-graphic characters, the single quote `'`, the double quote `"`, the question mark `?`,[21] and the backslash `\`, can be represented according to Table 9. The double quote `"` and the question mark `?`, can be represented as themselves or by the escape sequences `\"` and `\?` respectively, but the single quote `'`

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

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

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 9 are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

Table 9: Escape sequences [tab:lex.ccon.esc]

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*	`\x`*`hhh`*

7 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 `\x`*`hhh`* 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-literal*s with no prefix) or `wchar_t` (for *character-literal*s prefixed by `L`).

[*Note 4*: 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*]

8 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 5*: In translation phase 1, a *universal-character-name* is introduced whenever an actual 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 can use its own native character set, so long as the same results are obtained. — *end note*]

5.13.4 Floating-point literals [lex.fcon]

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

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

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

©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

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

sign: one of
  `+ -`

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

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

1 The type of a *floating-point-literal* is determined by its *floating-point-suffix* as specified in Table 10.

Table 10: Types of *floating-point-literal*s [tab:lex.fcon.type]

floating-point-suffix	type
none	`double`
`f` or `F`	`float`
`l` or `L`	`long double`

2 The *significand* of a *floating-point-literal* is the *fractional-constant* or *digit-sequence* of a *decimal-floating-point-literal* or the *hexadecimal-fractional-constant* or *hexadecimal-digit-sequence* of a *hexadecimal-floating-point-literal*. In the significand, the sequence of *digit*s or *hexadecimal-digit*s and optional period are interpreted as a base N real number s, where N is 10 for a *decimal-floating-point-literal* and 16 for a *hexadecimal-floating-point-literal*.

[*Note 1*: Any optional separating single quotes are ignored when determining the value. — *end note*]

If an *exponent-part* or *binary-exponent-part* is present, the exponent e of the *floating-point-literal* is the result of interpreting the sequence of an optional *sign* and the *digit*s as a base 10 integer. Otherwise, the exponent e is 0. The scaled value of the literal is $s \times 10^e$ for a *decimal-floating-point-literal* and $s \times 2^e$ for a *hexadecimal-floating-point-literal*.

[*Example 1*: The *floating-point-literal*s `49.625` and `0xC.68p+2` have the same value. The *floating-point-literal*s `1.602'176'565e-19` and `1.602176565e-19` have the same value. — *end example*]

3 If the scaled value is not in the range of representable values for its type, the program is ill-formed. Otherwise, the value of a *floating-point-literal* is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner.

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 basic 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

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

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.

1 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.

2 [*Note 1*: The characters `'('` and `')'` are permitted in a *raw-string*. Thus, `R"delimiter((a|b))delimiter"` is equivalent to `"(a|b)"`. — *end note*]

3 [*Note 2*: 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*]

4 [*Example 1*: 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*]

5 After translation phase 6, a *string-literal* that does not begin with an *encoding-prefix* is an *ordinary string literal*. An ordinary string literal has type "array of *n* `const char`" where *n* is the size of the string as defined below, has static storage duration (6.7.5), and is initialized with the given characters.

6 A *string-literal* that begins with `u8`, such as `u8"asdf"`, is a *UTF-8 string literal*. A UTF-8 string literal has type "array of *n* `const char8_t`", where *n* is the size of the string as defined below; each successive element of the object representation (6.8) has the value of the corresponding code unit of the UTF-8 encoding of the string.

7 Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals.

8 A *string-literal* that begins with `u`, such as `u"asdf"`, is a *UTF-16 string literal*. A UTF-16 string literal has type "array of *n* `const char16_t`", where *n* is the size of the string as defined below; each successive element of the array has the value of the corresponding code unit of the UTF-16 encoding of the string.

[*Note 3*: A single *c-char* may produce more than one `char16_t` character in the form of surrogate pairs. A surrogate pair is a representation for a single code point as a sequence of two 16-bit code units. — *end note*]

9 A *string-literal* that begins with `U`, such as `U"asdf"`, is a *UTF-32 string literal*. A UTF-32 string literal has type "array of *n* `const char32_t`", where *n* is the size of the string as defined below; each successive element of the array has the value of the corresponding code unit of the UTF-32 encoding of the string.

10 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.

11 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 concatenations are conditionally-supported with implementation-defined behavior.

[*Note 4*: 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 11 has some examples of valid concatenations.

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

Table 11: String literal concatenations [tab:lex.string.concat]

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 2*:

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

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

12 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.

13 Escape sequences and *universal-character-name*s in non-raw string literals have the same meaning as in *character-literal*s (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 UTF-16 string literal may yield a surrogate pair. In a narrow string literal, a *universal-character-name* may map to more than one `char` or `char8_t` 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 UTF-16 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 5*: The size of a `char16_t` string literal is the number of code units, not the number of characters. — *end note*]

[*Note 6*: Any *universal-character-name*s are required to correspond to a code point in the range [0, D800) or [E000, 10FFFF] (hexadecimal) (5.3). — *end note*]

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'`.

14 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-literal*s 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 7*: 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`

1 The pointer literal is the keyword `nullptr`. It is a prvalue of type `std::nullptr_t`.

[*Note 1*: `std::nullptr_t` is a distinct type that is neither a pointer type nor a pointer-to-member type; rather, a prvalue of this type is a null pointer constant and can be converted to a null pointer value or null member pointer value. See 7.3.12 and 7.3.13. — *end note*]

5.13.8 User-defined literals [lex.ext]

user-defined-literal:
user-defined-integer-literal
user-defined-floating-point-literal
user-defined-string-literal
user-defined-character-literal

©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

user-defined-integer-literal:
  decimal-literal ud-suffix
  octal-literal ud-suffix
  hexadecimal-literal ud-suffix
  binary-literal ud-suffix

user-defined-floating-point-literal:
  fractional-constant exponent-part$_{opt}$ *ud-suffix*
  digit-sequence exponent-part ud-suffix
  hexadecimal-prefix hexadecimal-fractional-constant binary-exponent-part ud-suffix
  hexadecimal-prefix hexadecimal-digit-sequence binary-exponent-part ud-suffix

user-defined-string-literal:
  string-literal ud-suffix

user-defined-character-literal:
  character-literal ud-suffix

ud-suffix:
  identifier

1 If a token matches both *user-defined-literal* and another *literal* kind, it is treated as the latter.

[*Example 1*: `123_km` is a *user-defined-literal*, but `12LL` is an *integer-literal*. — *end example*]

The syntactic non-terminal preceding the *ud-suffix* in a *user-defined-literal* is taken to be the longest sequence of characters that matches the syntax of that non-terminal.

2 A *user-defined-literal* is treated as a call to a literal operator or literal operator template (12.8). To determine the form of this call for a given *user-defined-literal* L with *ud-suffix* X, the *literal-operator-id* whose literal suffix identifier is X is looked up in the context of L using the rules for unqualified name lookup (6.5.2). Let S be the set of declarations found by this lookup. S shall not be empty.

3 If L is a *user-defined-integer-literal*, let n be the literal without its *ud-suffix*. If S contains a literal operator with parameter type `unsigned long long`, the literal L is treated as a call of the form

```
operator "" X(nULL)
```

Otherwise, S shall contain a raw literal operator or a numeric literal operator template (12.8) but not both. If S contains a raw literal operator, the literal L is treated as a call of the form

```
operator "" X("n")
```

Otherwise (S contains a numeric literal operator template), L is treated as a call of the form

```
operator "" X<'c1', 'c2', ... 'ck'>()
```

where n is the source character sequence $c_1c_2...c_k$.

[*Note 1*: The sequence $c_1c_2...c_k$ can only contain characters from the basic source character set. — *end note*]

4 If L is a *user-defined-floating-point-literal*, let f be the literal without its *ud-suffix*. If S contains a literal operator with parameter type `long double`, the literal L is treated as a call of the form

```
operator "" X(fL)
```

Otherwise, S shall contain a raw literal operator or a numeric literal operator template (12.8) but not both. If S contains a raw literal operator, the *literal* L is treated as a call of the form

```
operator "" X("f")
```

Otherwise (S contains a numeric literal operator template), L is treated as a call of the form

```
operator "" X<'c1', 'c2', ... 'ck'>()
```

where f is the source character sequence $c_1c_2...c_k$.

[*Note 2*: The sequence $c_1c_2...c_k$ can only contain characters from the basic source character set. — *end note*]

5 If L is a *user-defined-string-literal*, let *str* be the literal without its *ud-suffix* and let *len* be the number of code units in *str* (i.e., its length excluding the terminating null character). If S contains a literal operator template with a non-type template parameter for which *str* is a well-formed *template-argument*, the literal L is treated as a call of the form

```
operator "" X<str>()
```

Otherwise, the literal L is treated as a call of the form

```
operator "" X(str, len)
```


©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

6 If L is a *user-defined-character-literal*, let *ch* be the literal without its *ud-suffix*. S shall contain a literal operator (12.8) whose only parameter has the type of *ch* and the literal L is treated as a call of the form

```
operator "" X(ch)
```

7 [*Example 2*:

```
long double operator "" _w(long double);
std::string operator "" _w(const char16_t*, std::size_t);
unsigned operator "" _w(const char*);
int main() {
  1.2_w;            // calls operator "" _w(1.2L)
  u"one"_w;         // calls operator "" _w(u"one", 3)
  12_w;             // calls operator "" _w("12")
  "two"_w;          // error: no applicable literal operator
}
```

— *end example*]

8 In translation phase 6 (5.2), adjacent *string-literal*s are concatenated and *user-defined-string-literal*s are considered *string-literal*s for that purpose. During concatenation, *ud-suffix*es are removed and ignored and the concatenation process occurs as described in 5.13.5. At the end of phase 6, if a *string-literal* is the result of a concatenation involving at least one *user-defined-string-literal*, all the participating *user-defined-string-literal*s shall have the same *ud-suffix* and that suffix is applied to the result of the concatenation.

9 [*Example 3*:

```
int main() {
  L"A" "B" "C"_x;    // OK: same as L"ABC"_x
  "P"_x "Q" "R"_y;   // error: two different ud-suffixes
}
```

— *end example*]

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

6 Basics [basic]

6.1 Preamble [basic.pre]

1 [*Note 1*: This Clause presents the basic concepts of the C++ language. It explains the difference between an object and a name and how they relate to the value categories for expressions. It introduces the concepts of a declaration and a definition and presents C++'s notion of type, scope, linkage, and storage duration. The mechanisms for starting and terminating a program are discussed. Finally, this Clause presents the fundamental types of the language and lists the ways of constructing compound types from these. — *end note*]

2 [*Note 2*: This Clause does not cover concepts that affect only a single part of the language. Such concepts are discussed in the relevant Clauses. — *end note*]

3 An *entity* is a value, object, reference, structured binding, function, enumerator, type, class member, bit-field, template, template specialization, namespace, or pack.

4 A *name* is a use of an *identifier* (5.10), *operator-function-id* (12.6), *literal-operator-id* (12.8), *conversion-function-id* (11.4.8.3), or *template-id* (13.3) that denotes an entity or label (8.7.6, 8.2).

5 Every name that denotes an entity is introduced by a *declaration*. Every name that denotes a label is introduced either by a `goto` statement (8.7.6) or a *labeled-statement* (8.2).

6 A *variable* is introduced by the declaration of a reference other than a non-static data member or of an object. The variable's name, if any, denotes the reference or object.

7 A *local entity* is a variable with automatic storage duration (6.7.5.4), a structured binding (9.6) whose corresponding variable is such an entity, or the `*this` object (7.5.2).

8 Some names denote types or templates. In general, whenever a name is encountered it is necessary to determine whether that name denotes one of these entities before continuing to parse the program that contains it. The process that determines this is called *name lookup* (6.5).

9 Two names are *the same* if

(9.1) — they are *identifier*s composed of the same character sequence, or

(9.2) — they are *operator-function-id*s formed with the same operator, or

(9.3) — they are *conversion-function-id*s formed with the same type, or

(9.4) — they are *template-id*s that refer to the same class, function, or variable (13.6), or

(9.5) — they are *literal-operator-id*s (12.8) formed with the same literal suffix identifier.

10 A name used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (6.6) of the name specified in each translation unit.

6.2 Declarations and definitions [basic.def]

1 A declaration (Clause 9) may introduce one or more names into a translation unit or redeclare names introduced by previous declarations. If so, the declaration specifies the interpretation and semantic properties of these names. A declaration may also have effects including:

(1.1) — a static assertion (9.1),

(1.2) — controlling template instantiation (13.9.3),

(1.3) — guiding template argument deduction for constructors (13.7.2.3),

(1.4) — use of attributes (9.12), and

(1.5) — nothing (in the case of an *empty-declaration*).

2 Each entity declared by a *declaration* is also *defined* by that declaration unless:

(2.1) — it declares a function without specifying the function's body (9.5),

(2.2) — it contains the `extern` specifier (9.2.2) or a *linkage-specification*[22] (9.11) and neither an *initializer* nor a *function-body*,

22) Appearing inside the brace-enclosed *declaration-seq* in a *linkage-specification* does not affect whether a declaration is a definition.

 ©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

(2.3) — it declares a non-inline static data member in a class definition (11.4, 11.4.9),

(2.4) — it declares a static data member outside a class definition and the variable was defined within the class with the `constexpr` specifier (this usage is deprecated; see D.7),

(2.5) — it is introduced by an *elaborated-type-specifier* (11.3),

(2.6) — it is an *opaque-enum-declaration* (9.7.1),

(2.7) — it is a *template-parameter* (13.2),

(2.8) — it is a *parameter-declaration* (9.3.4.6) in a function declarator that is not the *declarator* of a *function-definition*,

(2.9) — it is a `typedef` declaration (9.2.4),

(2.10) — it is an *alias-declaration* (9.2.4),

(2.11) — it is a *using-declaration* (9.9),

(2.12) — it is a *deduction-guide* (13.7.2.3),

(2.13) — it is a *static_assert-declaration* (9.1),

(2.14) — it is an *attribute-declaration* (9.1),

(2.15) — it is an *empty-declaration* (9.1),

(2.16) — it is a *using-directive* (9.8.4),

(2.17) — it is a *using-enum-declaration* (9.7.2),

(2.18) — it is a *template-declaration* (13.1) whose *template-head* is not followed by either a *concept-definition* or a *declaration* that defines a function, a class, a variable, or a static data member.

(2.19) — it is an explicit instantiation declaration (13.9.3), or

(2.20) — it is an explicit specialization (13.9.4) whose *declaration* is not a definition.

A declaration is said to be a *definition* of each entity that it defines.

[*Example 1*: All but one of the following are definitions:

```
int a;                          // defines a
extern const int c = 1;         // defines c
int f(int x) { return x+a; }    // defines f and defines x
struct S { int a; int b; };     // defines S, S::a, and S::b
struct X {                      // defines X
  int x;                        // defines non-static data member x
  static int y;                 // declares static data member y
  X(): x(0) { }                 // defines a constructor of X
};
int X::y = 1;                   // defines X::y
enum { up, down };              // defines up and down
namespace N { int d; }          // defines N and N::d
namespace N1 = N;               // defines N1
X anX;                          // defines anX
```

whereas these are just declarations:

```
extern int a;                   // declares a
extern const int c;             // declares c
int f(int);                     // declares f
struct S;                       // declares S
typedef int Int;                // declares Int
extern X anotherX;              // declares anotherX
using N::d;                     // declares d
```

— *end example*]

3 [*Note 1*: In some circumstances, C++ implementations implicitly define the default constructor (11.4.5.2), copy constructor, move constructor (11.4.5.3), copy assignment operator, move assignment operator (11.4.6), or destructor (11.4.7) member functions. — *end note*]

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

[*Example 2*: Given

```
#include <string>

struct C {
  std::string s;                    // std::string is the standard library class (21.3)
};

int main() {
  C a;
  C b = a;
  b = a;
}
```

the implementation will implicitly define functions to make the definition of `C` equivalent to

```
struct C {
  std::string s;
  C() : s() { }
  C(const C& x): s(x.s) { }
  C(C&& x): s(static_cast<std::string&&>(x.s)) { }
      //  : s(std::move(x.s)) { }
  C& operator=(const C& x) { s = x.s; return *this; }
  C& operator=(C&& x) { s = static_cast<std::string&&>(x.s); return *this; }
      //              { s = std::move(x.s); return *this; }
  ~C() { }
};
```

— *end example*]

4 [*Note 2*: A class name can also be implicitly declared by an *elaborated-type-specifier* (9.2.9.4). — *end note*]

5 In the definition of an object, the type of that object shall not be an incomplete type (6.8), an abstract class type (11.7.4), or a (possibly multi-dimensional) array thereof.

6.3 One-definition rule [basic.def.odr]

1 No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, template, default argument for a parameter (for a function in a given scope), or default template argument.

2 An expression or conversion is *potentially evaluated* unless it is an unevaluated operand (7.2), a subexpression thereof, or a conversion in an initialization or conversion sequence in such a context. The set of *potential results* of an expression E is defined as follows:

(2.1) — If E is an *id-expression* (7.5.4), the set contains only E.

(2.2) — If E is a subscripting operation (7.6.1.2) with an array operand, the set contains the potential results of that operand.

(2.3) — If E is a class member access expression (7.6.1.5) of the form E_1 `.` $\texttt{template}_{opt}$ E_2 naming a non-static data member, the set contains the potential results of E_1.

(2.4) — If E is a class member access expression naming a static data member, the set contains the *id-expression* designating the data member.

(2.5) — If E is a pointer-to-member expression (7.6.4) of the form E_1 `.*` E_2, the set contains the potential results of E_1.

(2.6) — If E has the form `(`E_1`)`, the set contains the potential results of E_1.

(2.7) — If E is a glvalue conditional expression (7.6.16), the set is the union of the sets of potential results of the second and third operands.

(2.8) — If E is a comma expression (7.6.20), the set contains the potential results of the right operand.

(2.9) — Otherwise, the set is empty.

[*Note 1*: This set is a (possibly-empty) set of *id-expressions*, each of which is either E or a subexpression of E.

[*Example 1*: In the following example, the set of potential results of the initializer of `n` contains the first `S::x` subexpression, but not the second `S::x` subexpression.

```
struct S { static const int x = 0; };
```


©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

```
const int &f(const int &r);
int n = b ? (1, S::x)          // S::x is not odr-used here
          : f(S::x);           // S::x is odr-used here, so a definition is required
```

— *end example*]

— *end note*]

3 A function is *named by* an expression or conversion as follows:

(3.1) — A function is named by an expression or conversion if it is the selected member of an overload set (6.5, 12.4, 12.5) in an overload resolution performed as part of forming that expression or conversion, unless it is a pure virtual function and either the expression is not an *id-expression* naming the function with an explicitly qualified name or the expression forms a pointer to member (7.6.2.2).

[*Note 2*: This covers taking the address of functions (7.3.4, 7.6.2.2), calls to named functions (7.6.1.3), operator overloading (Clause 12), user-defined conversions (11.4.8.3), allocation functions for *new-expressions* (7.6.2.8), as well as non-default initialization (9.4). A constructor selected to copy or move an object of class type is considered to be named by an expression or conversion even if the call is actually elided by the implementation (11.10.6). — *end note*]

(3.2) — A deallocation function for a class is named by a *new-expression* if it is the single matching deallocation function for the allocation function selected by overload resolution, as specified in 7.6.2.8.

(3.3) — A deallocation function for a class is named by a *delete-expression* if it is the selected usual deallocation function as specified in 7.6.2.9 and 11.12.

4 A variable `x` whose name appears as a potentially-evaluated expression E is *odr-used* by E unless

(4.1) — `x` is a reference that is usable in constant expressions (7.7), or

(4.2) — `x` is a variable of non-reference type that is usable in constant expressions and has no mutable subobjects, and E is an element of the set of potential results of an expression of non-volatile-qualified non-class type to which the lvalue-to-rvalue conversion (7.3.2) is applied, or

(4.3) — `x` is a variable of non-reference type, and E is an element of the set of potential results of a discarded-value expression (7.2) to which the lvalue-to-rvalue conversion is not applied.

5 A structured binding is odr-used if it appears as a potentially-evaluated expression.

6 `*this` is odr-used if `this` appears as a potentially-evaluated expression (including as the result of the implicit transformation in the body of a non-static member function (11.4.3)).

7 A virtual member function is odr-used if it is not pure. A function is odr-used if it is named by a potentially-evaluated expression or conversion. A non-placement allocation or deallocation function for a class is odr-used by the definition of a constructor of that class. A non-placement deallocation function for a class is odr-used by the definition of the destructor of that class, or by being selected by the lookup at the point of definition of a virtual destructor (11.4.7).[23]

8 An assignment operator function in a class is odr-used by an implicitly-defined copy-assignment or move-assignment function for another class as specified in 11.4.6. A constructor for a class is odr-used as specified in 9.4. A destructor for a class is odr-used if it is potentially invoked (11.4.7).

9 A local entity (6.1) is *odr-usable* in a declarative region (6.4.1) if:

(9.1) — either the local entity is not `*this`, or an enclosing class or non-lambda function parameter scope exists and, if the innermost such scope is a function parameter scope, it corresponds to a non-static member function, and

(9.2) — for each intervening declarative region (6.4.1) between the point at which the entity is introduced and the region (where `*this` is considered to be introduced within the innermost enclosing class or non-lambda function definition scope), either:

(9.2.1) — the intervening declarative region is a block scope, or

(9.2.2) — the intervening declarative region is the function parameter scope of a *lambda-expression* that has a *simple-capture* naming the entity or has a *capture-default*, and the block scope of the *lambda-expression* is also an intervening declarative region.

If a local entity is odr-used in a declarative region in which it is not odr-usable, the program is ill-formed.

23) An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.

©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

[*Example 2*:

```
void f(int n) {
  [] { n = 1; };                  // error: n is not odr-usable due to intervening lambda-expression
  struct A {
    void f() { n = 2; }           // error: n is not odr-usable due to intervening function definition scope
  };
  void g(int = n);                // error: n is not odr-usable due to intervening function parameter scope
  [=](int k = n) {};              // error: n is not odr-usable due to being
                                  // outside the block scope of the lambda-expression
  [&] { [n]{ return n; }; };      // OK
}
```

— *end example*]

10 Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement (8.5.2); no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 11.4.5.2, 11.4.5.3, 11.4.7, and 11.4.6).

[*Example 3*:

```
auto f() {
  struct A {};
  return A{};
}
decltype(f()) g();
auto x = g();
```

A program containing this translation unit is ill-formed because `g` is odr-used but not defined, and cannot be defined in any other translation unit because the local class `A` cannot be named outside this translation unit. — *end example*]

11 A *definition domain* is a *private-module-fragment* or the portion of a translation unit excluding its *private-module-fragment* (if any). A definition of an inline function or variable shall be reachable from the end of every definition domain in which it is odr-used outside of a discarded statement.

12 A definition of a class shall be reachable in every context in which the class is used in a way that requires the class type to be complete.

[*Example 4*: The following complete translation unit is well-formed, even though it never defines `X`:

```
struct X;                         // declare X as a struct type
struct X* x1;                     // use X in pointer formation
X* x2;                            // use X in pointer formation
```

— *end example*]

[*Note 3*: The rules for declarations and expressions describe in which contexts complete class types are required. A class type `T` must be complete if:

(12.1) — an object of type `T` is defined (6.2), or

(12.2) — a non-static class data member of type `T` is declared (11.4), or

(12.3) — `T` is used as the allocated type or array element type in a *new-expression* (7.6.2.8), or

(12.4) — an lvalue-to-rvalue conversion is applied to a glvalue referring to an object of type `T` (7.3.2), or

(12.5) — an expression is converted (either implicitly or explicitly) to type `T` (7.3, 7.6.1.4, 7.6.1.7, 7.6.1.9, 7.6.3), or

(12.6) — an expression that is not a null pointer constant, and has type other than *cv* `void*`, is converted to the type pointer to `T` or reference to `T` using a standard conversion (7.3), a `dynamic_cast` (7.6.1.7) or a `static_cast` (7.6.1.9), or

(12.7) — a class member access operator is applied to an expression of type `T` (7.6.1.5), or

(12.8) — the `typeid` operator (7.6.1.8) or the `sizeof` operator (7.6.2.5) is applied to an operand of type `T`, or

(12.9) — a function with a return type or argument type of type `T` is defined (6.2) or called (7.6.1.3), or

(12.10) — a class with a base class of type `T` is defined (11.7), or

(12.11) — an lvalue of type `T` is assigned to (7.6.19), or

(12.12) — the type `T` is the subject of an `alignof` expression (7.6.2.6), or

(12.13) — an *exception-declaration* has type `T`, reference to `T`, or pointer to `T` (14.4).

— *end note*]

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

13 There can be more than one definition of a

(13.1) — class type (Clause 11),

(13.2) — enumeration type (9.7.1),

(13.3) — inline function or variable (9.2.8),

(13.4) — templated entity (13.1),

(13.5) — default argument for a parameter (for a function in a given scope) (9.3.4.7), or

(13.6) — default template argument (13.2)

in a program provided that each definition appears in a different translation unit and the definitions satisfy the following requirements. Given such an entity `D` defined in more than one translation unit, for all definitions of `D`, or, if `D` is an unnamed enumeration, for all definitions of `D` that are reachable at any given program point, the following requirements shall be satisfied.

(13.7) — Each such definition shall not be attached to a named module (10.1).

(13.8) — Each such definition shall consist of the same sequence of tokens, where the definition of a closure type is considered to consist of the sequence of tokens of the corresponding *lambda-expression*.

(13.9) — In each such definition, corresponding names, looked up according to 6.5, shall refer to the same entity, after overload resolution (12.4) and after matching of partial template specialization (13.10.4), except that a name can refer to

(13.9.1) — a non-volatile const object with internal or no linkage if the object

(13.9.1.1) — has the same literal type in all definitions of `D`,

(13.9.1.2) — is initialized with a constant expression (7.7),

(13.9.1.3) — is not odr-used in any definition of `D`, and

(13.9.1.4) — has the same value in all definitions of `D`,

or

(13.9.2) — a reference with internal or no linkage initialized with a constant expression such that the reference refers to the same entity in all definitions of `D`.

(13.10) — In each such definition, except within the default arguments and default template arguments of `D`, corresponding *lambda-expression*s shall have the same closure type (see below).

(13.11) — In each such definition, corresponding entities shall have the same language linkage.

(13.12) — In each such definition, the overloaded operators referred to, the implicit calls to conversion functions, constructors, operator new functions and operator delete functions, shall refer to the same function.

(13.13) — In each such definition, a default argument used by an (implicit or explicit) function call or a default template argument used by an (implicit or explicit) *template-id* or *simple-template-id* is treated as if its token sequence were present in the definition of `D`; that is, the default argument or default template argument is subject to the requirements described in this paragraph (recursively).

(13.14) — If `D` is a class with an implicitly-declared constructor (11.4.5.2, 11.4.5.3), it is as if the constructor was implicitly defined in every translation unit where it is odr-used, and the implicit definition in every translation unit shall call the same constructor for a subobject of `D`.

[*Example 5*:

```
// translation unit 1:
struct X {
  X(int, int);
  X(int, int, int);
};
X::X(int, int = 0) { }
class D {
  X x = 0;
};
D d1;                          // X(int, int) called by D()
```


©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

```
// translation unit 2:
struct X {
  X(int, int);
  X(int, int, int);
};
X::X(int, int = 0, int = 0) { }
class D {
  X x = 0;
};
D d2;                          // X(int, int, int) called by D();
                               // D()'s implicit definition violates the ODR
```

— *end example*]

(13.15) — If D is a class with a defaulted three-way comparison operator function (11.11.3), it is as if the operator was implicitly defined in every translation unit where it is odr-used, and the implicit definition in every translation unit shall call the same comparison operators for each subobject of D.

14 If D is a template and is defined in more than one translation unit, then the preceding requirements shall apply both to names from the template's enclosing scope used in the template definition (13.8.4), and also to dependent names at the point of instantiation (13.8.3). These requirements also apply to corresponding entities defined within each definition of D (including the closure types of *lambda-expression*s, but excluding entities defined within default arguments or default template arguments of either D or an entity not defined within D). For each such entity and for D itself, the behavior is as if there is a single entity with a single definition, including in the application of these requirements to other entities.

[*Note 4*: The entity is still declared in multiple translation units, and 6.6 still applies to these declarations. In particular, *lambda-expression*s (7.5.5) appearing in the type of D can result in the different declarations having distinct types, and *lambda-expression*s appearing in a default argument of D can still denote different types in different translation units. — *end note*]

15 If these definitions do not satisfy these requirements, then the program is ill-formed; a diagnostic is required only if the entity is attached to a named module and a prior definition is reachable at the point where a later definition occurs.

16 [*Example 6*:

```
inline void f(bool cond, void (*p)()) {
  if (cond) f(false, []{});
}
inline void g(bool cond, void (*p)() = []{}) {
  if (cond) g(false);
}
struct X {
  void h(bool cond, void (*p)() = []{}) {
    if (cond) h(false);
  }
};
```

If the definition of f appears in multiple translation units, the behavior of the program is as if there is only one definition of f. If the definition of g appears in multiple translation units, the program is ill-formed (no diagnostic required) because each such definition uses a default argument that refers to a distinct *lambda-expression* closure type. The definition of X can appear in multiple translation units of a valid program; the *lambda-expression*s defined within the default argument of X::h within the definition of X denote the same closure type in each translation unit. — *end example*]

17 If, at any point in the program, there is more than one reachable unnamed enumeration definition in the same scope that have the same first enumerator name and do not have typedef names for linkage purposes (9.7.1), those unnamed enumeration types shall be the same; no diagnostic required.

6.4 Scope [basic.scope]

6.4.1 Declarative regions and scopes [basic.scope.declarative]

1 Every name is introduced in some portion of program text called a *declarative region*, which is the largest part of the program in which that name is valid, that is, in which that name may be used as an unqualified name to refer to the same entity. In general, each particular name is valid only within some possibly discontiguous portion of program text called its *scope*. To determine the scope of a declaration, it is sometimes convenient to refer to the *potential scope* of a declaration. The scope of a declaration is the same as its potential scope

 ©ISO/IEC 2020 – All rights reserved

Documents Document Sharing Hub

Full document details: BS ISO/IEC 14882:2020

unless the potential scope contains another declaration of the same name. In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.

2 [*Example 1*: In

```
int j = 24;
int main() {
  int i = j, j;
  j = 42;
}
```

the identifier `j` is declared twice as a name (and used twice). The declarative region of the first `j` includes the entire example. The potential scope of the first `j` begins immediately after that `j` and extends to the end of the program, but its (actual) scope excludes the text between the `,` and the `}`. The declarative region of the second declaration of `j` (the `j` immediately before the semicolon) includes all the text between `{` and `}`, but its potential scope excludes the declaration of `i`. The scope of the second declaration of `j` is the same as its potential scope. — *end example*]

3 The names declared by a declaration are introduced into the scope in which the declaration occurs, except that the presence of a `friend` specifier (11.9.4), certain uses of the *elaborated-type-specifier* (9.2.9.4), and *using-directives* (9.8.4) alter this general behavior.

4 Given a set of declarations in a single declarative region, each of which specifies the same unqualified name,

(4.1) — they shall all refer to the same entity, or all refer to functions and function templates; or

(4.2) — exactly one declaration shall declare a class name or enumeration name that is not a typedef name and the other declarations shall all refer to the same variable, non-static data member, or enumerator, or all refer to functions and function templates; in this case the class name or enumeration name is hidden (6.4.10).

[*Note 1*: A structured binding (9.6), namespace name (9.8), or class template name (13.1) must be unique in its declarative region. — *end note*]

[*Note 2*: These restrictions apply to the declarative region into which a name is introduced, which is not necessarily the same as the region in which the declaration occurs. In particular, *elaborated-type-specifiers* (9.2.9.4) and friend declarations (11.9.4) can introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to that region. Local extern declarations (6.6) can introduce a name into the declarative region where the declaration appears and also introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to both regions. — *end note*]

5 For a given declarative region R and a point P outside R, the set of *intervening* declarative regions between P and R comprises all declarative regions that are or enclose R and do not enclose P.

6 [*Note 3*: The name lookup rules are summarized in 6.5. — *end note*]

6.4.2 Point of declaration [basic.scope.pdecl]

1 The *point of declaration* for a name is immediately after its complete declarator (9.3) and before its *initializer* (if any), except as noted below.

[*Example 1*:

```
unsigned char x = 12;
{ unsigned char x = x; }
```

Here, the initialization of the second `x` has undefined behavior, because the initializer accesses the second `x` outside its lifetime (6.7.3). — *end example*]

2 [*Note 1*: A name from an outer scope remains visible up to the point of declaration of the name that hides it.

[*Example 2*:

```
const int  i = 2;
{ int  i[i]; }
```

declares a block-scope array of two integers. — *end example*]

— *end note*]

3 The point of declaration for a class or class template first declared by a *class-specifier* is immediately after the *identifier* or *simple-template-id* (if any) in its *class-head* (11.1). The point of declaration for an enumeration is immediately after the *identifier* (if any) in either its *enum-specifier* (9.7.1) or its first *opaque-enum-declaration* (9.7.1), whichever comes first. The point of declaration of an alias or alias template immediately follows the *defining-type-id* to which the alias refers.

 ©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020

4 The point of declaration of a *using-declarator* that does not name a constructor is immediately after the *using-declarator* (9.9).

5 The point of declaration for an enumerator is immediately after its *enumerator-definition*.

[*Example 3*:

```
const int x = 12;
{ enum { x = x }; }
```

Here, the enumerator `x` is initialized with the value of the constant `x`, namely 12. — *end example*]

6 After the point of declaration of a class member, the member name can be looked up in the scope of its class.

[*Note 2*: This is true even if the class is an incomplete class. For example,

```
struct X {
  enum E { z = 16 };
  int b[X::z];        // OK
};
```

— *end note*]

7 The point of declaration of a class first declared in an *elaborated-type-specifier* is as follows:

(7.1) — for a declaration of the form

class-key attribute-specifier-seq$_{opt}$ *identifier* ;

the *identifier* is declared to be a *class-name* in the scope that contains the declaration, otherwise

(7.2) — for an *elaborated-type-specifier* of the form

class-key identifier

if the *elaborated-type-specifier* is used in the *decl-specifier-seq* or *parameter-declaration-clause* of a function defined in namespace scope, the *identifier* is declared as a *class-name* in the namespace that contains the declaration; otherwise, except as a friend declaration, the *identifier* is declared in the smallest namespace or block scope that contains the declaration.

[*Note 3*: These rules also apply within templates. — *end note*]

[*Note 4*: Other forms of *elaborated-type-specifier* do not declare a new name, and therefore must refer to an existing *type-name*. See 6.5.5 and 9.2.9.4. — *end note*]

8 The point of declaration for an injected-class-name (11.1) is immediately following the opening brace of the class definition.

9 The point of declaration for a function-local predefined variable (9.5.1) is immediately before the *function-body* of a function definition.

10 The point of declaration of a structured binding (9.6) is immediately after the *identifier-list* of the structured binding declaration.

11 The point of declaration for the variable or the structured bindings declared in the *for-range-declaration* of a range-based `for` statement (8.6.5) is immediately after the *for-range-initializer*.

12 The point of declaration for a template parameter is immediately after its complete *template-parameter*.

[*Example 4*:

```
typedef unsigned char T;
template<class T
  = T                 // lookup finds the typedef name of unsigned char
  , T                 // lookup finds the template parameter
    N = 0> struct A { };
```

— *end example*]

13 [*Note 5*: Friend declarations refer to functions or classes that are members of the nearest enclosing namespace, but they do not introduce new names into that namespace (9.8.2.3). Function declarations at block scope and variable declarations with the `extern` specifier at block scope refer to declarations that are members of an enclosing namespace, but they do not introduce new names into that scope. — *end note*]

14 [*Note 6*: For point of instantiation of a template, see 13.8.5.1. — *end note*]

©ISO/IEC 2020 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2020