BS ISO/IEC 14882:2011

BSI Standards Publication

Information technology — Programming languages — C++

bsi.

...making excellence a habit.™

Document Sharing Hub

National foreword

This British Standard is the UK implementation of ISO/IEC 14882:2011. It supersedes BS ISO/IEC 14882:2003 which is withdrawn.

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

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

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

© BSI 2011

ISBN 978 0 580 63699 8

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 October 2011.

Amendments issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

BS ISO/IEC 14882:2011

INTERNATIONAL STANDARD

ISO/IEC 14882

Third edition
2011-09-01

Information technology — Programming languages — C++

Technologies de l'information — Langages de programmation — C++

Reference number
ISO/IEC 14882:2011(E)

Documents
Document Sharing Hub

© ISO/IEC 2011

Full document details: BS ISO/IEC 14882:2011

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2011

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

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

Contents

Contents iii

List of Tables xi

List of Figures xv

1 General 1
1.1 Scope . . . 1
1.2 Normative references . . . 1
1.3 Terms and definitions . . . 2
1.4 Implementation compliance . . . 5
1.5 Structure of this International Standard . . . 6
1.6 Syntax notation . . . 6
1.7 The C++ memory model . . . 7
1.8 The C++ object model . . . 7
1.9 Program execution . . . 8
1.10 Multi-threaded executions and data races . . . 12
1.11 Acknowledgments . . . 16

2 Lexical conventions 17
2.1 Separate translation . . . 17
2.2 Phases of translation . . . 17
2.3 Character sets . . . 18
2.4 Trigraph sequences . . . 19
2.5 Preprocessing tokens . . . 20
2.6 Alternative tokens . . . 21
2.7 Tokens . . . 21
2.8 Comments . . . 21
2.9 Header names . . . 22
2.10 Preprocessing numbers . . . 22
2.11 Identifiers . . . 22
2.12 Keywords . . . 23
2.13 Operators and punctuators . . . 24
2.14 Literals . . . 24

3 Basic concepts 34
3.1 Declarations and definitions . . . 34
3.2 One definition rule . . . 36
3.3 Scope . . . 38
3.4 Name lookup . . . 45
3.5 Program and linkage . . . 59
3.6 Start and termination . . . 62
3.7 Storage duration . . . 65
3.8 Object lifetime . . . 69
3.9 Types . . . 72
3.10 Lvalues and rvalues . . . 78

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

3.11 Alignment . . . 80

4 Standard conversions **81**
4.1 Lvalue-to-rvalue conversion . . . 82
4.2 Array-to-pointer conversion . . . 82
4.3 Function-to-pointer conversion . . . 82
4.4 Qualification conversions . . . 82
4.5 Integral promotions . . . 83
4.6 Floating point promotion . . . 84
4.7 Integral conversions . . . 84
4.8 Floating point conversions . . . 84
4.9 Floating-integral conversions . . . 85
4.10 Pointer conversions . . . 85
4.11 Pointer to member conversions . . . 85
4.12 Boolean conversions . . . 86
4.13 Integer conversion rank . . . 86

5 Expressions **87**
5.1 Primary expressions . . . 89
5.2 Postfix expressions . . . 97
5.3 Unary expressions . . . 109
5.4 Explicit type conversion (cast notation) . . . 117
5.5 Pointer-to-member operators . . . 118
5.6 Multiplicative operators . . . 119
5.7 Additive operators . . . 119
5.8 Shift operators . . . 121
5.9 Relational operators . . . 121
5.10 Equality operators . . . 122
5.11 Bitwise AND operator . . . 123
5.12 Bitwise exclusive OR operator . . . 123
5.13 Bitwise inclusive OR operator . . . 123
5.14 Logical AND operator . . . 123
5.15 Logical OR operator . . . 124
5.16 Conditional operator . . . 124
5.17 Assignment and compound assignment operators . . . 125
5.18 Comma operator . . . 127
5.19 Constant expressions . . . 127

6 Statements **130**
6.1 Labeled statement . . . 130
6.2 Expression statement . . . 130
6.3 Compound statement or block . . . 130
6.4 Selection statements . . . 131
6.5 Iteration statements . . . 133
6.6 Jump statements . . . 136
6.7 Declaration statement . . . 137
6.8 Ambiguity resolution . . . 138

7 Declarations **140**
7.1 Specifiers . . . 142
7.2 Enumeration declarations . . . 157

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

7.3 Namespaces . . . 161
7.4 The `asm` declaration . . . 173
7.5 Linkage specifications . . . 174
7.6 Attributes . . . 177

8 Declarators **182**
8.1 Type names . . . 183
8.2 Ambiguity resolution . . . 184
8.3 Meaning of declarators . . . 186
8.4 Function definitions . . . 198
8.5 Initializers . . . 202

9 Classes **216**
9.1 Class names . . . 218
9.2 Class members . . . 220
9.3 Member functions . . . 222
9.4 Static members . . . 225
9.5 Unions . . . 227
9.6 Bit-fields . . . 229
9.7 Nested class declarations . . . 229
9.8 Local class declarations . . . 231
9.9 Nested type names . . . 231

10 Derived classes **233**
10.1 Multiple base classes . . . 234
10.2 Member name lookup . . . 236
10.3 Virtual functions . . . 240
10.4 Abstract classes . . . 244

11 Member access control **246**
11.1 Access specifiers . . . 248
11.2 Accessibility of base classes and base class members . . . 249
11.3 Friends . . . 251
11.4 Protected member access . . . 254
11.5 Access to virtual functions . . . 255
11.6 Multiple access . . . 256
11.7 Nested classes . . . 256

12 Special member functions **257**
12.1 Constructors . . . 257
12.2 Temporary objects . . . 260
12.3 Conversions . . . 262
12.4 Destructors . . . 265
12.5 Free store . . . 267
12.6 Initialization . . . 269
12.7 Construction and destruction . . . 275
12.8 Copying and moving class objects . . . 278
12.9 Inheriting constructors . . . 286

13 Overloading **289**
13.1 Overloadable declarations . . . 289

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

13.2 Declaration matching . . . 291
13.3 Overload resolution . . . 292
13.4 Address of overloaded function . . . 311
13.5 Overloaded operators . . . 313
13.6 Built-in operators . . . 317

14 Templates **321**
14.1 Template parameters . . . 322
14.2 Names of template specializations . . . 325
14.3 Template arguments . . . 327
14.4 Type equivalence . . . 333
14.5 Template declarations . . . 334
14.6 Name resolution . . . 352
14.7 Template instantiation and specialization . . . 366
14.8 Function template specializations . . . 378

15 Exception handling **400**
15.1 Throwing an exception . . . 401
15.2 Constructors and destructors . . . 403
15.3 Handling an exception . . . 403
15.4 Exception specifications . . . 405
15.5 Special functions . . . 409

16 Preprocessing directives **411**
16.1 Conditional inclusion . . . 413
16.2 Source file inclusion . . . 414
16.3 Macro replacement . . . 415
16.4 Line control . . . 420
16.5 Error directive . . . 421
16.6 Pragma directive . . . 421
16.7 Null directive . . . 421
16.8 Predefined macro names . . . 421
16.9 Pragma operator . . . 423

17 Library introduction **424**
17.1 General . . . 424
17.2 The C standard library . . . 425
17.3 Definitions . . . 425
17.4 Additional definitions . . . 428
17.5 Method of description (Informative) . . . 428
17.6 Library-wide requirements . . . 434

18 Language support library **454**
18.1 General . . . 454
18.2 Types . . . 454
18.3 Implementation properties . . . 455
18.4 Integer types . . . 464
18.5 Start and termination . . . 465
18.6 Dynamic memory management . . . 467
18.7 Type identification . . . 473
18.8 Exception handling . . . 475

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

18.9 Initializer lists . . . 480
18.10 Other runtime support . . . 481

19 Diagnostics library **484**
19.1 General . . . 484
19.2 Exception classes . . . 484
19.3 Assertions . . . 488
19.4 Error numbers . . . 489
19.5 System error support . . . 489

20 General utilities library **500**
20.1 General . . . 500
20.2 Utility components . . . 500
20.3 Pairs . . . 504
20.4 Tuples . . . 508
20.5 Class template `bitset` . . . 518
20.6 Memory . . . 525
20.7 Smart pointers . . . 540
20.8 Function objects . . . 566
20.9 Metaprogramming and type traits . . . 585
20.10 Compile-time rational arithmetic . . . 602
20.11 Time utilities . . . 605
20.12 Class template `scoped_allocator_adaptor` . . . 620
20.13 Class `type_index` . . . 625

21 Strings library **628**
21.1 General . . . 628
21.2 Character traits . . . 628
21.3 String classes . . . 634
21.4 Class template `basic_string` . . . 638
21.5 Numeric conversions . . . 665
21.6 Hash support . . . 666
21.7 Null-terminated sequence utilities . . . 667

22 Localization library **671**
22.1 General . . . 671
22.2 Header `<locale>` synopsis . . . 671
22.3 Locales . . . 672
22.4 Standard `locale` categories . . . 684
22.5 Standard code conversion facets . . . 725
22.6 C library locales . . . 726

23 Containers library **728**
23.1 General . . . 728
23.2 Container requirements . . . 728
23.3 Sequence containers . . . 754
23.4 Associative containers . . . 786
23.5 Unordered associative containers . . . 803
23.6 Container adaptors . . . 819

24 Iterators library **829**

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

24.1 General . . . 829
24.2 Iterator requirements . . . 829
24.3 Header `<iterator>` synopsis . . . 834
24.4 Iterator primitives . . . 837
24.5 Iterator adaptors . . . 841
24.6 Stream iterators . . . 855

25 Algorithms library **863**
25.1 General . . . 863
25.2 Non-modifying sequence operations . . . 873
25.3 Mutating sequence operations . . . 878
25.4 Sorting and related operations . . . 887
25.5 C library algorithms . . . 900

26 Numerics library **902**
26.1 General . . . 902
26.2 Numeric type requirements . . . 902
26.3 The floating-point environment . . . 903
26.4 Complex numbers . . . 904
26.5 Random number generation . . . 914
26.6 Numeric arrays . . . 959
26.7 Generalized numeric operations . . . 981
26.8 C library . . . 984

27 Input/output library **989**
27.1 General . . . 989
27.2 Iostreams requirements . . . 990
27.3 Forward declarations . . . 990
27.4 Standard iostream objects . . . 992
27.5 Iostreams base classes . . . 994
27.6 Stream buffers . . . 1013
27.7 Formatting and manipulators . . . 1023
27.8 String-based streams . . . 1049
27.9 File-based streams . . . 1061

28 Regular expressions library **1076**
28.1 General . . . 1076
28.2 Definitions . . . 1076
28.3 Requirements . . . 1077
28.4 Header `<regex>` synopsis . . . 1079
28.5 Namespace `std::regex_constants` . . . 1086
28.6 Class `regex_error` . . . 1089
28.7 Class template `regex_traits` . . . 1089
28.8 Class template `basic_regex` . . . 1092
28.9 Class template `sub_match` . . . 1097
28.10 Class template `match_results` . . . 1103
28.11 Regular expression algorithms . . . 1108
28.12 Regular expression iterators . . . 1113
28.13 Modified ECMAScript regular expression grammar . . . 1119

29 Atomic operations library **1122**

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

29.1 General . . . 1122
29.2 Header `<atomic>` synopsis . . . 1122
29.3 Order and consistency . . . 1125
29.4 Lock-free property . . . 1128
29.5 Atomic types . . . 1128
29.6 Operations on atomic types . . . 1132
29.7 Flag type and operations . . . 1137
29.8 Fences . . . 1138

30 Thread support library **1140**
30.1 General . . . 1140
30.2 Requirements . . . 1140
30.3 Threads . . . 1143
30.4 Mutual exclusion . . . 1149
30.5 Condition variables . . . 1162
30.6 Futures . . . 1170

A Grammar summary **1187**
A.1 Keywords . . . 1187
A.2 Lexical conventions . . . 1187
A.3 Basic concepts . . . 1192
A.4 Expressions . . . 1192
A.5 Statements . . . 1195
A.6 Declarations . . . 1196
A.7 Declarators . . . 1200
A.8 Classes . . . 1202
A.9 Derived classes . . . 1203
A.10 Special member functions . . . 1203
A.11 Overloading . . . 1204
A.12 Templates . . . 1204
A.13 Exception handling . . . 1205
A.14 Preprocessing directives . . . 1205

B Implementation quantities **1207**

C Compatibility **1209**
C.1 C++ and ISO C . . . 1209
C.2 C++ and ISO C++ 2003 . . . 1218
C.3 C standard library . . . 1225

D Compatibility features **1229**
D.1 Increment operator with `bool` operand . . . 1229
D.2 `register` keyword . . . 1229
D.3 Implicit declaration of copy functions . . . 1229
D.4 Dynamic exception specifications . . . 1229
D.5 C standard library headers . . . 1229
D.6 Old iostreams members . . . 1230
D.7 `char*` streams . . . 1231
D.8 Function objects . . . 1240
D.9 Binders . . . 1243
D.10 `auto_ptr` . . . 1245

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

D.11 Violating *exception-specifications* 1247

E Universal character names for identifier characters **1249**
E.1 Ranges of characters allowed 1249
E.2 Ranges of characters disallowed initially 1249

F Cross references **1250**

Index **1268**

Index of grammar productions **1297**

Index of library names **1300**

Index of implementation-defined behavior **1336**

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

List of Tables

1 Trigraph sequences . . . 19
2 Alternative tokens . . . 21
3 Identifiers with special meaning . . . 23
4 Keywords . . . 23
5 Alternative representations . . . 24
6 Types of integer constants . . . 25
7 Escape sequences . . . 27
8 String literal concatenations . . . 30

9 Relations on `const` and `volatile` . . . 78

10 *simple-type-specifiers* and the types they specify . . . 154

11 Relationship between operator and function call notation . . . 297
12 Conversions . . . 305

13 Library categories . . . 424
14 C++ library headers . . . 435
15 C++ headers for C library facilities . . . 435
16 C++ headers for freestanding implementations . . . 436
17 `EqualityComparable` requirements . . . 437
18 `LessThanComparable` requirements . . . 437
19 `DefaultConstructible` requirements . . . 437
20 `MoveConstructible` requirements . . . 438
21 `CopyConstructible` requirements (in addition to MoveConstructible) . . . 438
22 `MoveAssignable` requirements . . . 438
23 `CopyAssignable` requirements(in addition to MoveAssignable) . . . 438
24 `Destructible` requirements . . . 438
25 `NullablePointer` requirements . . . 440
26 `Hash` requirements . . . 441
27 Descriptive variable definitions . . . 441
28 Allocator requirements . . . 442

29 Language support library summary . . . 454
30 Header `<cstddef>` synopsis . . . 454
31 Header `<climits>` synopsis . . . 464
32 Header `<cfloat>` synopsis . . . 464
33 Header `<cstdlib>` synopsis . . . 466
34 Header `<csetjmp>` synopsis . . . 482
35 Header `<csignal>` synopsis . . . 482
36 Header `<cstdalign>` synopsis . . . 482
37 Header `<cstdarg>` synopsis . . . 482
38 Header `<cstdbool>` synopsis . . . 482
39 Header `<cstdlib>` synopsis . . . 482
40 Header `<ctime>` synopsis . . . 483

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

41 Diagnostics library summary . . . 484
42 Header `<cassert>` synopsis . . . 488
43 Header `<cerrno>` synopsis . . . 489

44 General utilities library summary . . . 500
45 Header `<cstdlib>` synopsis . . . 539
46 Header `<cstring>` synopsis . . . 540
47 Primary type category predicates . . . 589
48 Composite type category predicates . . . 589
49 Type property predicates . . . 590
50 Type property queries . . . 595
51 Type relationship predicates . . . 596
52 Const-volatile modifications . . . 597
53 Reference modifications . . . 598
54 Sign modifications . . . 598
55 Array modifications . . . 599
56 Pointer modifications . . . 599
57 Other transformations . . . 600
58 Expressions used to perform ratio arithmetic . . . 604
59 Clock requirements . . . 608
60 Header `<ctime>` synopsis . . . 619

61 Strings library summary . . . 628
62 Character traits requirements . . . 629
63 `basic_string(const Allocator&)` effects . . . 643
64 `basic_string(const basic_string&)` effects . . . 643
65 `basic_string(const basic_string&, size_type, size_type, const Allocator&)` effects . 643
66 `basic_string(const charT*, size_type, const Allocator&)` effects . . . 644
67 `basic_string(const charT*, const Allocator&)` effects . . . 644
68 `basic_string(size_t, charT, const Allocator&)` effects . . . 644
69 `basic_string(const basic_string&, const Allocator&)` and `basic_string(basic_string&&, const Allocator&)` effects . . . 645
70 `operator=(const basic_string<charT, traits, Allocator>&)` effects . . . 645
71 `operator=(const basic_string<charT, traits, Allocator>&&)` effects . . . 645
72 `compare()` results . . . 659
73 Potential `mbstate_t` data races . . . 668
74 Header `<cctype>` synopsis . . . 668
75 Header `<cwctype>` synopsis . . . 669
76 Header `<cstring>` synopsis . . . 669
77 Header `<cwchar>` synopsis . . . 669
78 Header `<cstdlib>` synopsis . . . 669
79 Header `<cuchar>` synopsis . . . 670

80 Localization library summary . . . 671
81 Locale category facets . . . 675
82 Required specializations . . . 676
83 `do_in/do_out` result values . . . 694
84 `do_unshift` result values . . . 694
85 Integer conversions . . . 698
86 Length modifier . . . 698
87 Integer conversions . . . 702

List of Tables

© ISO/IEC 2011 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

88 Floating-point conversions . . . 703
89 Length modifier . . . 703
90 Numeric conversions . . . 703
91 Fill padding . . . 704
92 `do_get_date` effects . . . 711
93 Header `<clocale>` synopsis . . . 726
94 Potential `setlocale` data races . . . 727

95 Containers library summary . . . 728
96 Container requirements . . . 729
97 Reversible container requirements . . . 731
98 Optional container operations . . . 732
99 Allocator-aware container requirements . . . 733
100 Sequence container requirements (in addition to container) . . . 735
101 Optional sequence container operations . . . 737
102 Associative container requirements (in addition to container) . . . 740
103 Unordered associative container requirements (in addition to container) . . . 746

104 Iterators library summary . . . 829
105 Relations among iterator categories . . . 829
106 Iterator requirements . . . 831
107 Input iterator requirements (in addition to Iterator) . . . 831
108 Output iterator requirements (in addition to Iterator) . . . 832
109 Forward iterator requirements (in addition to input iterator) . . . 833
110 Bidirectional iterator requirements (in addition to forward iterator) . . . 833
111 Random access iterator requirements (in addition to bidirectional iterator) . . . 834

112 Algorithms library summary . . . 863
113 Header `<cstdlib>` synopsis . . . 900

114 Numerics library summary . . . 902
115 Seed sequence requirements . . . 915
116 Uniform random number generator requirements . . . 916
117 Random number engine requirements . . . 917
118 Random number distribution requirements . . . 921
119 Header `<cmath>` synopsis . . . 984
120 Header `<cstdlib>` synopsis . . . 985

121 Input/output library summary . . . 989
122 `fmtflags` effects . . . 999
123 `fmtflags` constants . . . 999
124 `iostate` effects . . . 999
125 `openmode` effects . . . 1000
126 `seekdir` effects . . . 1000
127 Position type requirements . . . 1004
128 `basic_ios::init()` effects . . . 1007
129 `basic_ios::copyfmt()` effects . . . 1008
130 `seekoff` positioning . . . 1054
131 `newoff` values . . . 1054
132 File open modes . . . 1064
133 `seekoff` effects . . . 1067

List of Tables

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

134 Header `<cstdio>` synopsis . . . 1074
135 Header `<cinttypes>` synopsis . . . 1075

136 Regular expressions library summary . . . 1076
137 Regular expression traits class requirements . . . 1077
138 `syntax_option_type` effects . . . 1087
139 `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first,last)`. . . . 1087
140 `error_type` values in the C locale . . . 1088
141 `match_results` assignment operator effects . . . 1105
142 Effects of `regex_match` algorithm . . . 1109
143 Effects of `regex_search` algorithm . . . 1110

144 Atomics library summary . . . 1122
145 `atomic` integral typedefs . . . 1131
146 `atomic <inttypes.h>` typedefs . . . 1132
147 Atomic arithmetic computations . . . 1136

148 Thread support library summary . . . 1140

149 Standard macros . . . 1225
150 Standard values . . . 1225
151 Standard types . . . 1226
152 Standard structs . . . 1226
153 Standard functions . . . 1226

154 C headers . . . 1229
155 `strstreambuf(streamsize)` effects . . . 1233
156 `strstreambuf(void* (*)(size_t), void (*)(void*))` effects . . . 1233
157 `strstreambuf(charT*, streamsize, charT*)` effects . . . 1234
158 `seekoff` positioning . . . 1236
159 `newoff` values . . . 1236

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

List of Figures

1 Expression category taxonomy 78

2 Directed acyclic graph 234
3 Non-virtual base 235
4 Virtual base 236
5 Virtual and non-virtual base 236
6 Name lookup 239

7 Stream position, offset, and size types [non-normative] 989

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

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.

ISO/IEC 14882 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This third edition cancels and replaces the second edition (ISO/IEC 14882:2003), which has been technically revised.

© ISO/IEC 2011 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

1 General [intro]

1.1 Scope [intro.scope]

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

2 C++ is a general purpose programming language based on the C programming language as specified in ISO/IEC 9899:1999, *Programming languages — C* (hereinafter referred to as the *C standard*). In addition to the facilities provided by C, C++ provides additional data types, classes, templates, exceptions, namespaces, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

1.2 Normative references [intro.refs]

1 The following referenced documents are indispensable for the application 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.

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

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

— ISO/IEC 9899:1999, *Programming languages — C*

— ISO/IEC 9899:1999/Cor.1:2001(E), *Programming languages — C, Technical Corrigendum 1*

— ISO/IEC 9899:1999/Cor.2:2004(E), *Programming languages — C, Technical Corrigendum 2*

— ISO/IEC 9899:1999/Cor.3:2007(E), *Programming languages — C, Technical Corrigendum 3*

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

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

— ISO/IEC TR 19769:2004, *Information technology — Programming languages, their environments and system software interfaces — Extensions for the programming language C to support new character data types*

2 The library described in Clause 7 of ISO/IEC 9899:1999 and Clause 7 of ISO/IEC 9899:1999/Cor.1:2001 and Clause 7 of ISO/IEC 9899:1999/Cor.2:2004 is hereinafter called the *C standard library*.[1]

3 The library described in ISO/IEC TR 19769:2004 is hereinafter called the *C Unicode TR*.

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

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

1) With the qualifications noted in Clauses 18 through 30 and in C.3, the C standard library is a subset of the C++ standard library.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

1.3 Terms and definitions [intro.defs]

1 For the purposes of this document, the following definitions apply.

2 17.3 defines additional terms that are used only in Clauses 17 through 30 and Annex D.

3 Terms that are used only in a small portion of this International Standard are defined where they are used and italicized where they are defined.

1.3.1 [defns.argument]
argument
actual argument
actual parameter
<function call expression> expression in the comma-separated list bounded by the parentheses

1.3.2 [defns.argument.macro]
argument
actual argument
actual parameter
<function-like macro> sequence of preprocessing tokens in the comma-separated list bounded by the parentheses

1.3.3 [defns.argument.throw]
argument
actual argument
actual parameter
<throw expression> the operand of `throw`

1.3.4 [defns.argument.templ]
argument
actual argument
actual parameter
<template instantiation> expression, *type-id* or *template-name* in the comma-separated list bounded by the angle brackets

1.3.5 [defns.cond.supp]
conditionally-supported
program construct that an implementation is not required to support
[*Note:* Each implementation documents all conditionally-supported constructs that it does not support. — *end note*]

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

1.3.7 [defns.dynamic.type]
dynamic type
<glvalue> type of the most derived object (1.8) to which the glvalue denoted by a glvalue expression refers

§ 1.3

© ISO/IEC 2011 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

[*Example:* if a pointer (8.3.1) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (Clause 10), the dynamic type of the expression `*p` is "`D`." References (8.3.2) are treated similarly. — *end example*]

1.3.8 **[defns.dynamic.type.prvalue]**
dynamic type
<prvalue> static type of the prvalue expression

1.3.9 **[defns.ill.formed]**
ill-formed program
program that is not well formed

1.3.10 **[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

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

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

1.3.13 **[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:* The extended character set is a superset of the basic character set (2.3). — *end note*]

1.3.14 **[defns.parameter]**
parameter
formal argument
formal 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

1.3.15 **[defns.parameter.macro]**
parameter
formal argument
formal parameter
<function-like macro> identifier from the comma-separated list bounded by the parentheses immediately following the macro name

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

1.3.16 **[defns.parameter.templ]**
parameter
formal argument
formal parameter
<template> *template-parameter*

1.3.17 **[defns.signature]**
signature
<function> name, parameter type list (8.3.5), and enclosing namespace (if any)
[*Note:* Signatures are used as a basis for name mangling and linking. — *end note*]

1.3.18 **[defns.signature.templ]**
signature
<function template> name, parameter type list (8.3.5), enclosing namespace (if any), return type, and template parameter list

1.3.19 **[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)

1.3.20 **[defns.signature.member]**
signature
<class member function> name, parameter type list (8.3.5), class of which the function is a member, *cv*-qualifiers (if any), and *ref-qualifier* (if any)

1.3.21 **[defns.signature.member.templ]**
signature
<class member function template> name, parameter type list (8.3.5), class of which the function is a member, *cv*-qualifiers (if any), *ref-qualifier* (if any), return type, and template parameter list

1.3.22 **[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)

1.3.23 **[defns.static.type]**
static type
type of an expression (3.9) resulting from analysis of the program without considering execution semantics
[*Note:* 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*]

1.3.24 **[defns.undefined]**
undefined behavior
behavior for which this International Standard imposes no requirements
[*Note:* Undefined behavior may be expected when this International Standard omits any explicit definition of

Documents
Document Sharing Hub

© ISO/IEC 2011 – All rights reserved

Full document details: BS ISO/IEC 14882:2011

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. — *end note*]

1.3.25 **[defns.unspecified]**
unspecified behavior
behavior, for a well-formed program construct and correct data, that depends on the implementation
[*Note:* The implementation is not required to document which behavior occurs. The range of possible behaviors is usually delineated by this International Standard. — *end note*]

1.3.26 **[defns.well.formed]**
well-formed program
C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).

1.4 Implementation compliance [intro.compliance]

1 The set of *diagnosable rules* consists of all syntactic and semantic rules in this International Standard 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 International Standard 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:

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

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

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

3 For classes and class templates, the library Clauses specify partial definitions. Private members (Clause 11) 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 (7.3). A C++ translation unit (2.2) obtains access to these names by including the appropriate standard library header (16.2).

6 The templates, classes, functions, and objects in the library have external linkage (3.5). The implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (2.2).

2) "Correct execution" can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

7 Two kinds of implementations are defined: a *hosted implementation* and a *freestanding implementation*. For a hosted implementation, this International Standard 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 (17.6.1.3).

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

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

1.5 Structure of this International Standard [intro.structure]

1 Clauses 2 through 16 describe the C++ programming language. That description includes detailed syntactic specifications in a form described in 1.6. For convenience, Annex A repeats all such syntactic specifications.

2 Clauses 18 through 30 and Annex D (the *library clauses*) describe the Standard C++ library. That description includes detailed descriptions of the templates, classes, functions, constants, and macros that constitute the library, in a form described in Clause 17.

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

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

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

1.6 Syntax notation [syntax]

1 In the syntax notation used in this International Standard, 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:

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

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

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

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

3) This documentation also defines implementation-defined behavior; see 1.9.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

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

1 The fundamental storage unit in the C++ memory model is the *byte*. A byte is at least large enough to contain any member of the basic execution character set (2.3) and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order bit*; the most significant bit is called the *high-order bit*. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

2 [*Note:* The representation of types is described in 3.9. — *end note*]

3 A *memory location* is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. [*Note:* Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. — *end note*] Two threads of execution (1.10) can update and access separate memory locations without interfering with each other.

4 [*Note:* Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields of non-zero width. — *end note*]

5 [*Example:* A structure declared as

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

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

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

1 The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is a region of storage. [*Note:* A function is not an object, regardless of whether or not it occupies storage in the way that objects do. — *end note*] An object is created by a *definition* (3.1), by a *new-expression* (5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a *name* (Clause 3). An object has a *storage duration* (3.7) which influences its *lifetime* (3.8). An object has a *type* (3.9). The term *object type* refers to the type with which the object is created. Some objects are *polymorphic* (10.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the *expressions* (Clause 5) used to access them.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

2 Objects can contain other objects, called *subobjects*. A subobject can be a *member subobject* (9.2), a *base class subobject* (Clause 10), or an array element. An object that is not a subobject of any other object is called a *complete object*.

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

— If `x` is a complete object, then `x` is the complete object of `x`.

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

4 If a complete object, a data member (9.2), or an array element is of class type, its type is considered the *most derived class*, to distinguish it from the class type of any base class subobject; an object of a most derived class type or of a non-class type is called a *most derived object*.

5 Unless it is a bit-field (9.6), a most derived object shall have a non-zero size and shall occupy one or more bytes of storage. Base class subobjects may have zero size. An object of trivially copyable or standard-layout type (3.9) shall occupy contiguous bytes of storage.

6 Unless an object is a bit-field or a base class subobject of zero size, the address of that object is the address of the first byte it occupies. Two objects that are not bit-fields may have the same address if one is a subobject of the other, or if at least one is a base class subobject of zero size and they are of different types; otherwise, they shall have distinct addresses.[4]

[*Example:*

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

— *end example*]

7 [*Note:* C++ provides a variety of fundamental types and several ways of composing new types from existing types (3.9). — *end note*]

1.9 Program execution [intro.execution]

1 The semantic descriptions in this International Standard define a parameterized nondeterministic abstract machine. This International Standard 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.[5]

2 Certain aspects and operations of the abstract machine are described in this International Standard 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.[6] 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 International Standard as unspecified (for example, order of evaluation of arguments to a function). Where possible, this International

4) Under the "as-if" rule an implementation is allowed to store two objects at the same machine address or not store an object at all if the program cannot observe the difference (1.9).

5) This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this International Standard 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.

6) This documentation also includes conditonally-supported constructs and locale-specific behavior. See 1.4.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

Standard 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 International Standard as undefined (for example, the effect of attempting to modify a `const` object). [*Note:* This International Standard 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 International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

6 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects which are neither

— of type `volatile std::sig_atomic_t` nor

— lock-free atomic objects (29.4)

are unspecified during the execution of the signal handler, and the value of any object not in either of these two categories that is modified by the handler becomes undefined.

7 An instance of each object with automatic storage duration (3.7.3) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

8 The least requirements on a conforming implementation are:

— Access to volatile objects are evaluated strictly according to the rules of the abstract machine.

— 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.

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

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

9 [*Note:* Operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative.[7] For example, in the following fragment

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

the expression statement behaves exactly the same as

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

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

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

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

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

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

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

or

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

since the values for `a` and `b` might have been, respectively, 4 and -8 or -17 and 12. However on a machine in which overflows do not produce an exception and in which the results of overflows are reversible, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur. — *end note*]

10 A *full-expression* is an expression that is not a subexpression of another expression. If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. A call to a destructor generated at the end of the lifetime of an object other than a temporary object is an implicit full-expression. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression.

[*Example:*

```
struct S {
  S(int i): I(i) { }
  int& v() { return I; }
private:
  int I;
};

 S s1(1);            // full-expression is call of S::S(int)
 S s2 = 2;           // full-expression is call of S::S(int)

void f() {
  if (S(3).v())      // full-expression includes lvalue-to-rvalue and
                     // int to bool conversions, performed before
                     // temporary is deleted at end of full-expression
  { }
}
```

— *end example*]

11 [*Note:* The evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. For example, subexpressions involved in evaluating default arguments (8.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — *end note*]

12 Accessing an object designated by a `volatile` glvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. *Evaluation* of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for glvalue evaluation and fetching a value previously assigned to an object for prvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access to a `volatile` object is evaluated the side effect is considered

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

complete, even though some external actions implied by the call (such as the I/O itself) or by the `volatile` access may not have completed yet.

13 *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread (1.10), which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. If A is not sequenced before B and B is not sequenced before A, then A and B are *unsequenced*. [*Note:* The execution of unsequenced evaluations can overlap. — *end note*] Evaluations A and B are *indeterminately sequenced* when either A is sequenced before B or B is sequenced before A, but it is unspecified which. [*Note:* Indeterminately sequenced evaluations cannot overlap, but either could be executed first. — *end note*]

14 Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.[8].

15 Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. [*Note:* In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. — *end note*] The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a scalar object is unsequenced relative to either anotherside effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined.

[*Example:*

```
void f(int, int);
void g(int i, int* v) {
  i = v[i++];          // the behavior is undefined
  i = 7, i++, i++;     // i becomes 9

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

  f(i = -1, i = -1);   // the behavior is undefined
}
```

— *end example*]

When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. [*Note:* Value computations and side effects associated with different argument expressions are unsequenced. — *end note*] Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[9] Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. [*Example:* Evaluation of a `new` expression invokes one or more allocation and constructor functions; see 5.3.4. For another example, invocation of a conversion function (12.3.2) can arise in contexts in which no function call syntax appears. — *end example*] The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

8) As specified in 12.2, after a full-expression is evaluated, a sequence of zero or more invocations of destructor functions for temporary objects takes place, usually in reverse order of the construction of each temporary object.

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

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

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

1 A *thread of execution* (also known as a *thread*) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread. [*Note:* When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread. — *end note*] Every thread in a program can potentially access every object and function in a program.[10] Under a hosted implementation, a C++ program can have more than one thread running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads. [*Note:* Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — *end note*] Under a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

2 Implementations should ensure that all unblocked threads eventually make progress. [*Note:* Standard library functions may silently block on I/O or locks. Factors in the execution environment, including externally-imposed thread priorities, may prevent an implementation from making certain guarantees of forward progress. — *end note*]

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

4 Two expression evaluations *conflict* if one of them modifies a memory location (1.7) and the other one accesses or modifies the same memory location.

5 The library defines a number of atomic operations (Clause 29) and operations on mutexes (Clause 30) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [*Note:* For example, a call that acquires a mutex will perform an acquire operation on the locations comprising the mutex. Correspondingly, a call that releases the same mutex will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on A. "Relaxed" atomic operations are not synchronization operations even though, like synchronization operations, they cannot contribute to data races. — *end note*]

6 All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M. If A and B are modifications of an atomic object M and A happens before (as defined below) B, then A shall precede B in the modification order of M, which is defined below. [*Note:* This states that the modification orders must respect the "happens before" relationship. — *end note*] [*Note:* There is a separate order for each atomic object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads may observe modifications to different objects in inconsistent orders. — *end note*]

10) An object with automatic or thread storage duration (3.7) is associated with one specific thread, and can be accessed by a different thread only indirectly through a pointer or reference (3.9.2).

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

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

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

— is an atomic read-modify-write operation.

8 Certain library calls *synchronize with* other library calls performed by another thread. For example, an atomic store-release synchronizes with a load-acquire that takes its value from the store (29.3). [*Note:* Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. — *end note*] [*Note:* The specifications of the synchronization operations define when one reads the value written by another. For atomic objects, the definition is clear. All operations on a given mutex occur in a single total order. Each mutex acquisition "reads the value written" by the last mutex release. — *end note*]

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

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

  — B is an invocation of any specialization of `std::kill_dependency` (29.3), or

  — A is the left operand of a built-in logical AND (`&&`, see 5.14) or logical OR (`||`, see 5.15) operator, or

  — A is the left operand of a conditional (`?:`, see 5.16) operator, or

  — A is the left operand of the built-in comma (`,`) operator (5.18);

  or

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

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

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

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

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

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

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

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

— A synchronizes with B, or

— A is dependency-ordered before B, or

— for some evaluation X

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

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

  — A inter-thread happens before X and X inter-thread happens before B.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

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

12 An evaluation A *happens before* an evaluation B if:

— A is sequenced before B, or

— A inter-thread happens before B.

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

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

— A happens before B and

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

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

14 The *visible sequence of side effects* on an atomic object M, with respect to a value computation B of M, is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first side effect is visible with respect to B, and for every side effect, it is not the case that B happens before it. The value of an atomic object M, as determined by evaluation B, shall be the value stored by some operation in the visible sequence of M with respect to B. [*Note:* It can be shown that the visible sequence of side effects of a value computation is unique given the coherence requirements below. — *end note*]

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

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

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

§ 1.10

© ISO/IEC 2011 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

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

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

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

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

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

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

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

— terminate,

— make a call to a library I/O function,

— access or modify a volatile object, or

— perform a synchronization operation or an atomic operation.

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

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

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

1.11 Acknowledgments [intro.ack]

1 The C++ programming language as described in this International Standard is based on the language as described in Chapter R (Reference Manual) of Stroustrup: *The C++ Programming Language* (second edition, Addison-Wesley Publishing Company, ISBN 0-201-53992-6, copyright ©1991 AT&T). That, in turn, is based on the C programming language as described in Appendix A of Kernighan and Ritchie: *The C Programming Language* (Prentice-Hall, 1978, ISBN 0-13-110163-3, copyright ©1978 AT&T).

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

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

4 All rights in these originals are reserved.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

2 Lexical conventions [lex]

2.1 Separate translation [lex.separate]

1 The text of the program is kept in units called *source files* in this International Standard. A source file together with all the headers (17.6.1.2) and source files included (16.2) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is called a *translation unit*. [*Note:* A C++ program need not all be translated at the same time. — *end note*]

2 [*Note:* Previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (3.5) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program (3.5). — *end note*]

2.2 Phases of translation [lex.phases]

1 The precedence among the syntax rules of translation is specified by the following phases.[11]

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. Trigraph sequences (2.4) are replaced by corresponding single-character internal representations. Any source file character not in the basic source character set (2.3) 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 (i.e., using the `\uXXXX` notation), are handled equivalently except where this replacement is reverted 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. If, as a result, a character sequence that matches the syntax of a universal-character-name is produced, 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 (2.5) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.[12] Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is unspecified. The process of dividing a source file's characters into preprocessing tokens is context-dependent. [*Example:* see the handling of `<` within a `#include` preprocessing directive. — *end example*]

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

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

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

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

5. Each source character set member 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 (2.14.3, 2.14.5); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[13]

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. (2.7). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [*Note:* The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (14.2). — *end note*] [*Note:* Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. — *end note*]

8. Translated translation units and instantiation units are combined as follows: [*Note:* Some or all of these may be supplied from a library. — *end note*] Each translated translation unit is examined to produce a list of required instantiations. [*Note:* This may include instantiations which have been explicitly requested (14.7.2). — *end note*] The definitions of the required templates are located. It is implementation-defined whether the source of the translation units containing these definitions is required to be available. [*Note:* An implementation could encode sufficient information into the translated translation unit so as to ensure the source is not required here. — *end note*] All the required instantiations are performed to produce *instantiation units*. [*Note:* These are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions. — *end note*] The program is ill-formed if any instantiation fails.

9. All external entity references are resolved. Library components are linked to satisfy external references to entities not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

2.3 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:[14]

```
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

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

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

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

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

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

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

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

The character designated by the universal-character-name `\UNNNNNNNN` is that character whose character short name in ISO/IEC 10646 is `NNNNNNNN`; the character designated by the universal-character-name `\uNNNN` is that character whose character short name in ISO/IEC 10646 is `0000NNNN`. If the hexadecimal value for a universal-character-name corresponds to a surrogate code point (in the range 0xD800–0xDFFF, inclusive), the program is ill-formed. Additionally, if the hexadecimal value for a universal-character-name outside the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of a character or string literal corresponds to a control character (in either of the ranges 0x00–0x1F or 0x7F–0x9F, both inclusive) or to a character in the basic source character set, the program is ill-formed.[15]

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 representation has all zero bits. 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.

2.4 Trigraph sequences [lex.trigraph]

1 Before any other processing takes place, each occurrence of one of the following sequences of three characters ("*trigraph sequences*") is replaced by the single character indicated in Table 1.

Table 1 — Trigraph sequences

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

2 [*Example:*

```
??=define arraycheck(a,b) a??(b??) ??!??! b??(a??)
```

becomes

```
#define arraycheck(a,b) a[b] || b[a]
```

— *end example*]

3 No other trigraph sequence exists. Each `?` that does not begin one of the trigraphs listed above is not changed.

15) A sequence of characters resembling a universal-character-name in an *r-char-sequence* (2.14.5) does not form a universal-character-name.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

2.5 Preprocessing tokens [lex.pptoken]

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

1 Each preprocessing token that is converted to a token (2.7) shall have the lexical form of a keyword, an identifier, a literal, an operator, or a punctuator.

2 A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: header names, identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories. If a `'` or a `"` character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by white space; this consists of comments (2.8), or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in Clause 16, 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:

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

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

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

— Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail.

[*Example:*

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

— *end example*]

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

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

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

2.6 Alternative tokens [lex.digraph]

1 Alternative token representations are provided for some operators and punctuators.[16]

2 In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.[17] The set of alternative tokens is defined in Table 2.

Table 2 — Alternative tokens

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

2.7 Tokens [lex.token]

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

1 There are five kinds of tokens: identifiers, keywords, literals,[18] 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:* Some white space is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. — *end note*]

2.8 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 with the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required. [*Note:* The comment characters

16) 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".

17) Thus the "stringized" values (16.3.2) of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

18) Literals include strings and character and numeric literals.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

`//`, `/*`, 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*]

2.9 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 Header name preprocessing tokens shall only appear within a `#include` preprocessing directive (16.2). 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 16.2.

2 The appearance of either of the characters `'` or `\` or of either of the character sequences `/*` or `//` in a *q-char-sequence* or an *h-char-sequence* is conditionally supported with implementation-defined semantics, as is the appearance of the character `"` in an *h-char-sequence*.[19]

2.10 Preprocessing numbers [lex.ppnumber]

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

1 Preprocessing number tokens lexically include all integral literal tokens (2.14.2) and all floating literal tokens (2.14.4).

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

2.11 Identifiers [lex.name]

identifier:
  identifier-nondigit
  identifier identifier-nondigit
  identifier digit

identifier-nondigit:
  nondigit
  universal-character-name
  other implementation-defined characters

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

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 14882:2011

nondigit: one of

```
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z _
```

digit: one of

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

1 An identifier is an arbitrarily long sequence of letters and digits. Each universal-character-name in an identifier shall designate a character whose encoding in ISO 10646 falls into one of the ranges specified in E.1. The initial element shall not be a universal-character-name designating a character whose encoding falls into one of the ranges specified in E.2. Upper- and lower-case letters are different. All characters are significant.[20]

2 The identifiers in Table 3 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. any ambiguity as to whether a given *identifier* has a special meaning is resolved to interpret the token as a regular *identifier*.

Table 3 — Identifiers with special meaning

override	final

3 In addition, some identifiers are reserved for use by C++ implementations and standard libraries (17.6.4.3.2) and shall not be used otherwise; no diagnostic is required.

2.12 Keywords [lex.key]

1 The identifiers shown in Table 4 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7) except in an *attribute-token* (7.6.1) [*Note:* The `export` keyword is unused but is reserved for future use. — *end note*]:

Table 4 — Keywords

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

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

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

2 Furthermore, the alternative representations shown in Table 5 for certain operators and punctuators (2.6) are reserved and shall not be used otherwise:

Table 5 — Alternative representations

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

2.13 Operators and punctuators [lex.operators]

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

preprocessing-op-or-punc: one of

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

Each *preprocessing-op-or-punc* is converted to a single token in translation phase 7 (2.2).

2.14 Literals [lex.literal]

2.14.1 Kinds of literals [lex.literal.kinds]

1 There are several kinds of literals.[21]

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

2.14.2 Integer literals [lex.icon]

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

decimal-literal:
- *nonzero-digit*
- *decimal-literal digit*

octal-literal:
- `0`
- *octal-literal octal-digit*

21) The term "literal" generally designates, in this International Standard, those tokens that are called "constants" in ISO C.

§ 2.14.2

© ISO/IEC 2011 – All rights reserved
Documents Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

hexadecimal-literal:
`0x` *hexadecimal-digit*
`0X` *hexadecimal-digit*
hexadecimal-literal hexadecimal-digit

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

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

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

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

unsigned-suffix: one of
`u U`

long-suffix: one of
`l L`

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

1 An *integer literal* is a sequence of digits that has no period or exponent part. An integer literal may have a prefix that specifies its base and a suffix that specifies its type. The lexically first digit of the sequence of digits is the most significant. A *decimal* integer literal (base ten) begins with a digit other than `0` and consists of a sequence of decimal digits. An *octal* integer literal (base eight) begins with the digit `0` and consists of a sequence of octal digits.[22] A *hexadecimal* integer literal (base sixteen) begins with `0x` or `0X` and consists of a sequence of hexadecimal digits, which include the decimal digits and the letters `a` through `f` and `A` through `F` with decimal values ten through fifteen. [*Example:* the number twelve can be written `12`, `014`, or `0XC`. — *end example*]

2 The type of an integer literal is the first of the corresponding list in Table 6 in which its value can be represented.

Table 6 — Types of integer constants

Suffix	Decimal constants	Octal or hexadecimal constant

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

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

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

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

§ 2.14.2

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

Table 6 — Types of integer constants (continued)

Suffix	Decimal constants	Octal or hexadecimal constant
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`

3 If an integer literal cannot be represented by any type in its list and an extended integer type (3.9.1) can represent its value, it may have that extended integer type. If all of the types in the list for the literal are signed, the extended integer type shall be signed. If all of the types in the list for the 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.

2.14.3 Character literals [lex.ccon]

character-literal:
  ' *c-char-sequence* '
  u' *c-char-sequence* '
  U' *c-char-sequence* '
  L' *c-char-sequence* '

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

c-char:
  any member of the source character set except
    the single-quote ', backslash \, or new-line character
  escape-sequence
  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 is one or more characters enclosed in single quotes, as in `'x'`, optionally preceded by one of the letters `u`, `U`, or `L`, as in `u'y'`, `U'z'`, or `L'x'`, respectively. A character literal that does not begin with `u`, `U`, or `L` is an ordinary character literal, also referred to as a narrow-character literal. An ordinary character literal that contains a single *c-char* has type `char`, with value equal to the numerical value of the

© ISO/IEC 2011 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

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 has type `int` and implementation-defined value.

2 A character literal that begins with the letter `u`, such as `u'y'`, is a character literal of type `char16_t`. The value of a `char16_t` literal containing a single *c-char* is equal to its ISO 10646 code point value, provided that the code point is representable with a single 16-bit code unit. (That is, provided it is a basic multi-lingual plane code point.) If the value is not representable within 16 bits, the program is ill-formed. A `char16_t` literal containing multiple *c-char*s is ill-formed. A character literal that begins with the letter `U`, such as `U'z'`, is a character literal of type `char32_t`. The value of a `char32_t` literal containing a single *c-char* is equal to its ISO 10646 code point value. A `char32_t` literal containing multiple *c-char*s is ill-formed. A character literal that begins with the letter `L`, such as `L'x'`, is a wide-character literal. A wide-character literal has type `wchar_t`.[23] The value of a wide-character literal containing a single *c-char* has value equal to the numerical value of the encoding of the *c-char* in the execution wide-character set, unless the *c-char* has no representation in the execution wide-character set, in which case the value is implementation-defined. [*Note:* The type `wchar_t` is able to represent all members of the execution wide-character set (see 3.9.1). — *end note*]. The value of a wide-character literal containing multiple *c-char*s is implementation-defined.

3 Certain nongraphic characters, the single quote `'`, the double quote `"`, the question mark `?`,[24] and the backslash `\`, can be represented according to Table 7. The double quote `"` and the question mark `?`, can be represented as themselves or by the escape sequences `\"` and `\?` respectively, but the single quote `'` and the backslash `\` shall be represented by the escape sequences `\'` and `\\` respectively. Escape sequences in which the character following the backslash is not listed in Table 7 are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

Table 7 — Escape sequences

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

4 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 literals with no prefix), `char16_t` (for literals prefixed by `'u'`), `char32_t` (for literals prefixed by `'U'`), or `wchar_t` (for literals prefixed by `'L'`).

23) They are intended for character sets where a character does not fit into a single byte.
24) Using an escape sequence for a question mark can avoid accidentally creating a trigraph.

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

5 A universal-character-name is translated to the encoding, in the appropriate execution character set, of the character named. If there is no such encoding, the universal-character-name is translated to an implementation-defined encoding. [*Note:* In translation phase 1, a universal-character-name is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in terms of universal-character-names. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained. — *end note*]

2.14.4 Floating literals [lex.fcon]

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

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

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

sign: one of
  `+ -`

digit-sequence:
  digit
  digit-sequence digit

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

1 A floating literal consists of an integer part, a decimal point, a fraction part, an `e` or `E`, an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (not both) can be omitted; either the decimal point or the letter `e` (or `E`) and the exponent (not both) can be omitted. The integer part, the optional decimal point and the optional fraction part form the *significant part* of the floating literal. The exponent, if present, indicates the power of 10 by which the significant part is to be scaled. If the scaled value is in the range of representable values for its type, the result is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner. The type of a floating literal is `double` unless explicitly specified by a suffix. The suffixes `f` and `F` specify `float`, the suffixes `l` and `L` specify `long double`. If the scaled value is not in the range of representable values for its type, the program is ill-formed.

2.14.5 String literals [lex.string]

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

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

s-char-sequence:
  s-char
  s-char-sequence s-char

§ 2.14.5

© ISO/IEC 2011 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

s-char:
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence
universal-character-name

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

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

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

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

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

1 A string literal is a sequence of characters (as defined in 2.14.3) surrounded by double quotes, optionally prefixed by R, u8, u8R, u, uR, U, UR, L, or LR, as in "...", R"(...)", u8"...", u8R"**(...)**", u"...", uR"*~(...)*~", U"...", UR"zzz(...)zzz", L"...", or LR"(...)", respectively.

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

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

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

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

— *end note*]

5 [*Example:* The raw string

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

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

```
R"(??)"
```

§ 2.14.5

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

is equivalent to `"\?\?"`. The raw string

```
R"#(
)??="
)#"
```

is equivalent to `"\n)\?\?=\"\n"`. — *end example*]

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

7 A string literal that begins with `u8`, such as `u8"asdf"`, is a UTF-8 string literal and is initialized with the given characters as encoded in UTF-8.

8 Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals. A narrow string literal has type "array of *n* `const char`", where *n* is the size of the string as defined below, and has static storage duration (3.7).

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

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

11 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 has static storage duration and is initialized with the given characters.

12 Whether all string literals are distinct (that is, are stored in nonoverlapping objects) is implementation-defined. The effect of attempting to modify a string literal is undefined.

13 In translation phase 6 (2.2), adjacent string literals are concatenated. If both string literals 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:* This concatenation is an interpretation, not a conversion. Because the interpretation happens in translation phase 6 (after each character from a 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 8 has some examples of valid concatenations.

Table 8 — String literal concatenations

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

Characters in concatenated strings are kept distinct.

[*Example:*

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

§ 2.14.5

© ISO/IEC 2011 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

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

14 After any necessary concatenation, in translation phase 7 (2.2), `'\0'` is appended to every string literal so that programs that scan a string can find its end.

15 Escape sequences and universal-character-names in non-raw string literals have the same meaning as in character literals (2.14.3), except that the single quote `'` is representable either by itself or by the escape sequence `\'`, and the double quote `"` shall be preceded by a `\`. In a narrow string literal, a universal-character-name may map to more than one `char` element due to *multibyte encoding*. The size of a `char32_t` or wide string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for the terminating `U'\0'` or `L'\0'`. The size of a `char16_t` string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for each character requiring a surrogate pair, plus one for the terminating `u'\0'`. [*Note:* The size of a `char16_t` string literal is the number of code units, not the number of characters. — *end note*] Within `char32_t` and `char16_t` literals, any universal-character-names shall be within the range `0x0` to `0x10FFFF`. The size of a narrow string literal is the total number of escape sequences and other characters, plus at least one for the multibyte encoding of each universal-character-name, plus one for the terminating `'\0'`.

2.14.6 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`.

2.14.7 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:* `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 4.10 and 4.11. — *end note*]

2.14.8 User-defined literals [lex.ext]

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

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

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

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

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

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011

ud-suffix:
identifier

1 If a token matches both *user-defined-literal* and another literal kind, it is treated as the latter. [*Example:* `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 could match that non-terminal.

2 A *user-defined-literal* is treated as a call to a literal operator or literal operator template (13.5.8). 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 (3.4.1). 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 (n ULL)
```

Otherwise, S shall contain a raw literal operator or a literal operator template (13.5.8) 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 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:* 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-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 (f L)
```

Otherwise, S shall contain a raw literal operator or a literal operator template (13.5.8) 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 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:* 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). The literal L is treated as a call of the form

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

6 If L is a *user-defined-character-literal*, let *ch* be the literal without its *ud-suffix*. S shall contain a literal operator (13.5.8) whose only parameter has the type *ch* and the literal L is treated as a call of the form

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

7 [*Example:*

§ 2.14.8

© ISO/IEC 2011 – All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 14882:2011