The New C Standard (C90 and C++)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

CHANGES

Commentary

The phrase *at the time of writing* is sometimes used. For this version of the material this time should be taken to mean no later than December 2008.

```
29 Jan 2008  1.1   Integrated in changes made by TC3, required C sentence renumbering.
                   60+ recent references added + associated commentary.
                   A few Usage figures and tables added.
                   Page layout improvements.  Lots of grammar fixes.
 5 Aug 2005  1.0b  Many hyperlinks added.  pdf searching through page 782 speeded up.
                   Various typos fixed (over 70% reported by Tom Plum).
16 Jun 2005  1.0a  Improvements to character set discussion (thanks to Kent Karlsson), margin
                   references, C99 footnote number typos, and various other typos fixed.
30 May 2005  1.0   Initial release.
```

Introduction

0 With the introduction of new devices and extended character sets, new features may be added to this International Standard. Subclauses in the language and library clauses warn implementors and programmers of usages which, though valid in themselves, may conflict with future additions.

Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this International Standard. They are retained because of their widespread use, but their use in new implementations (for implementation features) or new programs (for language [6.11] or library features [7.26]) is discouraged.

This International Standard is divided into four major subdivisions:

— preliminary elements (clauses 1–4);

— the characteristics of environments that translate and execute C programs (clause 5);

— the language syntax, constraints, and semantics (clause 6);

— the library facilities (clause 7).

Examples are provided to illustrate possible forms of the constructions described. Footnotes are provided to emphasize consequences of the rules described in that subclause or elsewhere in this International Standard. References are used to refer to other related subclauses. Recommendations are provided to give advice or guidance to implementors. Annexes provide additional information and summarize the information contained in this International Standard. A bibliography lists documents that were referred to during the preparation of the standard.

The language clause (clause 6) is derived from "The C Reference Manual".

The library clause (clause 7) is based on the 1984 */usr/group Standard*.

C90

C90 was the first version of the C Standard, known as ISO/IEC 9899:1990(E) (Ritchie[3] gives a history of prestandard development). It has now been officially superseded by C99. The C90 sections ask the question: What are the differences, if any, between the C90 Standard and the new C99 Standard?

Text such this occurs (with a bar in the margin) when a change of wording can lead to a developer visible change in behavior of a program.

Possible differences include:

- C90 said X was black, C99 says X is white.
- C99 has relaxed a requirement specified in C90.
- C99 has tightened a requirement specified in C90.
- C99 contains a construct that was not supported in C90.

If a construct is new in C99 this fact is only pointed out in the first sentence of any paragraph discussing it. This section is omitted if the wording is identical (word for word, or there are minor word changes that do not change the semantics) to that given in C99. Sometimes sentences have remained the same but have changed their location in the document. Such changes have not been highlighted.

The first C Standard was created by the US ANSI Committee X3J11 (since renamed as NCITS J11). This X3J11
document is sometimes called C89 after its year of publication as an ANSI standard (The shell and utilities portion of POSIX[2] specifies a c89 command, even although this standard references the ISO C Standard, not the ANSI one.). The published document was known as ANSI X3.159–1989.

This ANSI standard document was submitted, in 1990, to ISO for ratification as an International Standard. Some minor editorial changes needed to be made to the document to accommodate ISO rules (a sed script was used to make the changes to the troff sources from which the camera-ready copy of the ANSI and ISO standards was created). For instance, the word Standard was replaced by International Standard and some major section numbers were changed. More significantly, the Rationale ceased to be included as part of the document (and the list of names of the committee members was removed). After publication of this ISO standard in 1990, ANSI went through its procedures for withdrawing their original document and adopting the ISO Standard. Subsequent purchasers of the ANSI standard see, for instance, the words International Standard not just Standard.

1 Updates to C90

defect report

Part of the responsibility of an ISO Working Group is to provide answers to queries raised against any published standard they are responsible for. During the early 1990s, the appropriate ISO procedure seemed to be the one dealing with defects, and it was decided to create a Defect Report log (entries are commonly known as *DR*s). These procedures were subsequently updated and defect reports were renamed *interpretation requests* by ISO. The C committee continues to use the term *defect* and DR, as well as the new term *interpretation request*.

Standards Committees try to work toward a publication schedule. As the (self-imposed) deadline for publication of the C Standard grew nearer, several issues remained outstanding. Rather than delay the publication date, it was agreed that these issues should be the subject of an Amendment to the Standard. The purpose of this Amendment was to address issues from Denmark (readable trigraphs), Japan (additional support for wide character handling), and the UK (tightening up the specification of some constructs whose wording was considered to be ambiguous). The title of the Amendment was *C Integrity*.

As work on DRs (this is how they continue to be referenced in the official WG14 log) progressed, it became apparent that the issues raised by the UK, to be handled by the Amendment, were best dealt with via these same procedures. It was agreed that the UK work item would be taken out of the Amendment and converted into a series of DRs. The title of the Amendment remained the same even though the material that promoted the choice of title was no longer included within it.

To provide visibility for those cases in which a question had uncovered problems with wording in the published standard the Committee decided to publish collections of DRs. The ISO document containing such corrections is known as a Technical Corrigendum (*TC*) and two were published for C90. A TC is normative and contains edits to the existing standard's wording only, not the original question or any rationale behind the decision reached. An alternative to a TC is a Record of Response (*RR*), a non-normative document.

Wording from the Amendment, the TCs and decisions on defect reports that had not been formally published were integrated into the body of the C99 document.

A determined group of members of X3J11, the ANSI Committee, felt that C could be made more attractive to numerical programmers. To this end it was agreed that this Committee should work toward producing a technical report dealing with numerical issues.

NCEG

The Numerical C Extensions Group (*NCEG*) was formed on May 10, 1989; its official designation was X3J11.1. The group was disbanded on January 4, 1994. The group produced a number of internal, committee reports, but no officially recognized Technical Reports were produced. Topics covered included: compound literals and designation initializers, extended integers via a header, complex arithmetic, restricted pointers, variable length arrays, data parallel C extensions (a considerable amount of time was spent on discussing the merits of different approaches), and floating-point C extensions. Many of these reports were used as the base documents for constructs introduced into C99.

base document

Support for parallel threads of execution was not addressed by NCEG because there was already an ANSI Committee, X3H5, working toward standardizing a parallelism model and Fortran and C language bindings to it.

C++

Many developers view C++ as a superset of C and expect to be able to migrate C code to C++. While this book does not get involved in discussing the major redesigns that are likely to be needed to make effective use of C++, it does do its best to dispel the myth of C being a subset of C++. There may be a language that is common to both, but these sections tend to concentrate on the issues that need to be considered when translating C source using a C++ translator.

What does the C++ Standard, ISO/IEC 14882:1998(E), have to say about constructs that are in C99?

- *Wording is identical.* Say no more.
- *Wording is similar.* Slight English grammar differences, use of terminology differences and other minor issues. These are sometimes pointed out.
- *Wording is different but has the same meaning.* The sequence of words is too different to claim they are the same. But the meaning appears to be the same. These are not pointed out unless they highlight a C++ view of the world that is different from C.
- *Wording is different and has a different meaning.* Here the C++ wording is quoted, along with a discussion of the differences.
- *No C++ sentence can be associated with a C99 sentence.* This often occurs because of a construct that does not appear in the C++ Standard and this has been pointed out in a previous sentence occurring before this derived sentence.

There is a stylized form used to comment source code associated with C— `/* behavior */`— and C++— `// behavior`.

The precursor to C++ was known as C with Classes. While it was being developed C++ existed in an environment where there was extensive C expertise and C source code. Attempts by Stroustrup to introduce incompatibilities were met by complaints from his users.[4]

The intertwining of C and C++, in developers mind-sets, in vendors shipping a single translator with a language selection option, and in the coexistence of translation units written in either language making up one program means that it is necessary to describe any differences between the two.

The April 1989 meeting of WG14 was asked two questions by ISO: (1) should the C++ language be standardized, and (2) was WG14 the Committee that should do the work? The decision on (1) was very close, some arguing that C++ had not yet matured sufficiently to warrant being standardized, others arguing that working toward a standard would stabilize the language (constant changes to its specification and implementation were causing headaches for developers using it for mission-critical applications). Having agreed that there should be a C++ Standard WG14 was almost unanimous in stating that they were not the Committee that should create the standard. During April 1991 WG21, the ISO C++ Standard's Committee was formed; they met for the first time two months later.

In places additional background information on C++ is provided. Particularly where different concepts, or terminology, are used to describe what is essentially the same behavior.

In a few places constructs available in C++, but not C, are described. The rationale for this is that a C developer, only having a C++ translator to work with, might accidentally use a C++ construct. Many C++ translators offer a C compatibility mode, which often does little more than switch off support for a few C++ constructs. This description may also provide some background about why things are different in C++.

Everybody has a view point, even the creator of C++, Bjarne Stroustrup. But the final say belongs to the standards' body that oversees the development of language standards, SC22. The following was the initial position.

Resolutions Prepared at the Plenary Meeting of

ISO/IEC JTC 1/SC22

Vienna, Austria

September 23–29, 1991

Resolution AK Differences between C and C++

Notwithstanding that C and C++ are separate languages, ISO/IEC JTC1/SC22 directs WG21 to document differences in accordance with ISO/IEC TR 10176.

Resolution AL WG14 (C) and WG21 (C++) Coordination

While recognizing the need to preserve the respective and different goals of C and C++, ISO/IEC JTC1/SC22 directs WG14 and WG21 to ensure, in current and future development of their respective languages, that differences between C and C++ are kept to the minimum. The word "differences" is taken to refer to strictly conforming programs of C which either are invalid programs in C++ or have different semantics in C++.

This position was updated after work on the first C++ Standard had been completed, but too late to have any major impact on the revision of the C Standard.

Resolutions Prepared at the Eleventh Plenary Meeting of

ISO/IEC JTC 1/SC22

Snekkersten, Denmark

August 24–27, 1998

Resolution 98-6: Relationship Between the Work of WG21 and that of WG14

Recognizing that the user communities of the C and C++ languages are becoming increasingly divergent, ISO/IEC JTC 1/SC22 authorizes WG21 to carry out future revisions of ISO/IEC 14882:1998 (Programming Language C++) without necessarily adopting new C language features contained in the current revision to ISO/IEC 9899:1990 (Programming Language C) or any future revisions thereof.

ISO/IEC JTC 1/SC22 encourages WG14 and WG21 to continue their close cooperation in the future.

1. Scope

---

standard specifies form and interpretation

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

C++

1.1p1 *This International Standard specifies requirements for implementations of the C++ programming language.*

The C++ Standard does not specify the behavior of programs, but of implementations. For this standard the behavior of C++ programs has to be deduced from this, implementation-oriented, specification.

In those cases where the same wording is used in both standards, there is the potential for a different interpretation. In the case of the preprocessor, an entire clause has been copied, almost verbatim, from one document into the other. Given the problems that implementors are having producing a translator that handles the complete C++ Standard, and the pressures of market forces, it might be some time before people become interested in these distinctions.

It specifies 2

C++

The C++ Standard does not list the items considered to be within its scope.

— the syntax and constraints of the C language; 4

C++

1.1p1

The first such requirement is that they implement the language, and so this International Standard also defines C++.

While the specification of the C++ Standard includes syntax, it does not define and use the term *constraints*. What the C++ specification contains are *diagnosable rules*. A conforming implementation is required to check and issue a diagnostic if violated.

146 diagnostic shall produce

---

5 — the semantic rules for interpreting C programs;

C++

The C++ Standard specifies rules for implementations, not programs.

---

6 — the representation of input data to be processed by C programs;

C++

The C++ Standard is silent on this issue.

---

8 — the restrictions and limits imposed by a conforming implementation of C. limits specify

C90

The model of the minimal host expected to be able to translate a C program was assumed to have 64 K of free memory.

C++

Annex B contains an informative list of implementation limits. However, the C++ Standard does not specify any minimum limits that a conforming implementation must meet.

---

9 This International Standard does not specify

C++

The C++ Standard does not list any issues considered to be outside of its scope.

---

14 1) This International Standard is designed to promote the portability of C programs among a variety of data-processing systems. footnote 1

C++

No intended purpose is stated by the C++ Standard.

2. Normative references

---

19 For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. Dated references

C90

This sentence did not appear in the C90 Standard.

C++

At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. 1.2p1

The C++ Standard does not explicitly state whether later versions of standards do or do not apply. In the case of the C++ library, clause 17.3.1.4 refers to “the ISO C standard,”, which could be read to imply that agreements based on the C++ Standard may reference either the C90 library or the C99 library. The C++ ISO Technical Report TR 19768 (C++ Library Extensions) includes support for the wide character library functionality that is new in C99, but does not include support for some of the type generic maths functions (some of these are the subject of work on a separate TR) or extended integer types. However, the current C++ Standard document effective references the C90 library.

However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. 20

C90

This sentence did not appear in the C90 Standard.

For undated references, the latest edition of the normative document referred to applies. 21

C90

This sentence did not appear in the C90 Standard.

ISO 646 ISO/IEC 646, *Information technology— ISO 7-bit coded character set for information interchange.* 24

C++

1.2p1 *ISO/IEC 10646–1:1993 Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*

ISO 2382 ISO/IEC 2382–1:1993, *Information technology— Vocabulary — Part 1: Fundamental terms.* 25

C++

1.2p1 *ISO/IEC 2382 (all parts), Information technology — Vocabulary*

ISO 4217, *Codes for the representation of currencies and funds.* 26

C++

There is no mention of this document in the C++ Standard.

ISO 8601 ISO 8601, *Data elements and interchange formats— Information interchange— Representation of dates and times.* 27

C++

There is no mention of this document in the C++ Standard.

ISO 10646 ISO/IEC 10646 (all parts), *Information technology— Universal Multiple-Octet Coded Character Set (UCS).* 28

C++

1.2p1

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

ISO/IEC 10646:2003 is not divided into parts and the C++ Standard encourages the possibility of applying the most recent editions of standards.

19 Dated references

---

29 IEC 60559:1989, *Binary floating-point arithmetic for microprocessor systems* (previously designated IEC 559:1989). IEC 60559

C90

This standard did not specify a particular floating-point format, although the values given as an example for `<float.h>` were IEEE-754 specific (which is now an International Standard, IEC 60559).

C++

There is no mention of this document in the C++ Standard.

3. Terms, definitions, and symbols

C90

The title used in the C90 Standard was "Definitions and conventions".

---

30 For the purposes of this International Standard, the following definitions apply.

C++

For the purposes of this International Standard, the definitions given in ISO/IEC 2382 and the following definitions apply. 1.3p1

The following subclauses describe the definitions (17.1), and method of description (17.3) for the library. Clause 17.4 and clauses 18 through 27 specify the contents of the library, and library requirements and constraints on both well-formed C++ programs and conforming implementations. 17p9

---

31 Other terms are defined where they appear in *italic* type or on the left side of a syntax rule. terms defined where

C90

The fact that terms are defined when they appear "on the left side of a syntax rule" was not explicitly specified in the C90 Standard.

---

33 Terms not defined in this International Standard are to be interpreted according to ISO/IEC 2382–1.

C++

For the purposes of this International Standard, the definitions given in ISO/IEC 2382 and the following definitions apply. 1.3p1

The C++ Standard thus references all parts of the above standard. Not just the first part.

3.1

access

access 35

⟨execution-time action⟩ to read or modify the value of an object

C++

In C++ the term *access* is used primarily in the sense of *accessibility*; that is, the semantic rules dealing with when identifiers declared in different classes and namespaces can be referred to. The C++ Standard has a complete clause (Clause 11, Member access control) dealing with this issue. While the C++ Standard also uses *access* in the C sense (e.g., in 1.8p1), this is not the primary usage.

3.2

alignment

alignment 39

requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

C++

3.9p5 *Object types have alignment requirements (3.9.1, 3.9.2). The alignment of a complete object type is an implementation-defined integer value representing a number of bytes; an object is allocated at an address that meets the alignment requirements of its object type.*

There is no requirement on a C implementation to document its alignment requirements.

3.3

argument

argument 40

actual argument

actual parameter (deprecated)

expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

C++

The C++ definition, 1.3.1p1, does not include the terms *actual argument* and *actual parameter*.

3.4

3.4.1

implementation-defined behavior

implementation-defined behavior 42

unspecified behavior where each implementation documents how the choice is made

C90

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

The C99 wording has explicitly made the association between the terms implementation-defined and unspecified that was only implicit within the wording of the C90 Standard. It is possible to interpret the C90 definition as placing few restrictions on what an implementation-defined behavior might be. For instance, raising a signal or terminating program execution appear to be permitted. The C99 definition limits the possible behaviors to one of the possible behaviors permitted by the standard.

signed integer conversion implementation-defined 685

C++

The C++ Standard uses the same wording, 1.3.5, as C90.

3.4.2

3.4.3

46 **undefined behavior** undefined behavior

behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

C90

> *Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, or of indeterminately valued objects, for which this International Standard imposes no requirements.*

Use of an indeterminate value need not result in undefined behavior. If the value is read from an object that has **`unsigned char`** type, the behavior is unspecified. This is because objects of type **`unsigned char`** are required to represent values using a notation that does not support a trap representation.

75 indeterminate value
579 trap representation reading is undefined behavior
571 unsigned char pure binary

3.4.4

49 **unspecified behavior** unspecified behavior

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

C90

> *Behavior, for a correct program construct and correct data, for which this International Standard imposes no requirements.*

The C99 wording more clearly describes the intended meaning of the term *unspecified behavior*, given the contexts in which it is used.

C++

> *behavior, for a well-formed program construct and correct data, that depends on the implementation. The implementation is not required to document which behavior occurs. [Note: usually, the range of possible behaviors is delineated by this International Standard.]* 1.3.13

This specification suggests that there exist possible unspecified behaviors that are not delineated by the standard, while the wording in the C Standard suggests that all possible unspecified behaviors are mentioned.

3.5

51 **bit** bit

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

C++

ISO 2382 25 The C++ Standard does not explicitly define this term. The definition given in ISO 2382 is “Either of the digits 0 or 1 when used in the binary numeration system.”.

3.6

3.7

character <abstract>

character 58

⟨abstract⟩ member of a set of elements used for the organization, control, or representation of data

C++

The C++ Standard does not define the term *character*; however, it does reference ISO/IEC 2382. Part 1, Clause 01.02.11, defines *character* using very similar wording to that given in the C Standard. The following might also be considered applicable.

17.1.2 *in clauses 21, 22, and 27, means any object which, when treated sequentially, can represent text. The term does not only mean* **`char`** *and* **`wchar_t`** *objects, but any value that can be represented by a type that provides the definitions provided in these clauses.*

3.7.1

character single-byte

character 59

single-byte character ⟨C⟩ bit representation that fits in a byte

C++

The C++ Standard does not define the term *character*. This term has different meanings in different application contexts and human cultures. In a language that supports overloading, it makes no sense to restrict the usage of this term to a particular instance.

3.7.2

3.7.3

wide character

wide character 62

bit representation that fits in an object of type **`wchar_t`**, capable of representing any character in the current locale

C++

The C++ Standard uses the term *wide-character literal* and *wide-character sequences*, 17.3.2.1.3.3, but does not define the term *wide character*.

2.13.2p2 *A character literal that begins with the letter* `L`*, such as* `L'x'`*, is a wide-character literal.*

3.8

constraint

constraint 63

restriction, either syntactic or semantic, by which the exposition of language elements is to be interpreted

C++

The C++ Standard does not contain any constraints; it contains diagnosable rules.

> *a C++ program constructed according to the syntax rules, diagnosable semantics rules, and the One Definition Rule (3.2).* 1.3.14 well-formed program

However, the library does use the term constraints.

> *This subclause describes restrictions on C++ programs that use the facilities of the C++ Standard Library.* 17.4.3 Constraints on programs

> *This subclause describes the constraints upon, and latitude of, implementations of the C++ Standard library.* 17.4.4p1

But they are not constraints in the C sense of requiring a diagnostic to be issued if they are violated. Like C, the C++ Standard does not require any diagnostics to be issued during program execution.

3.9

---

64 **correctly rounded result** correctly rounded result

representation in the result format that is nearest in value, subject to the ~~effective~~current rounding mode, to what the result would be given unlimited range and precision

C++

This term is not defined in the C++ Standard (the term *rounded* only appears once, when discussing rounding toward zero).

3.10

3.11

---

66 **forward reference** forward reference

reference to a later subclause of this International Standard that contains additional information relevant to this subclause

C++

C++ does not contain any forward reference clauses. However, the other text in the other clauses contain significantly more references than C99 does.

3.12

---

67 **implementation** implementation

particular set of software, running in a particular translation environment under particular control options, that performs translation of programs for, and supports execution of functions in, a particular execution environment

C++

The C++ Standard does not provide a definition of what an implementation might be.

3.13

3.14

---

69 **object** object

region of data storage in the execution environment, the contents of which can represent values

C++

While an *object* is also defined, 1.8p1, to be a region of storage in C++, the term has many other connotations in an object-oriented language.

---

reference object

NOTE When referenced, an object may be interpreted as having a particular type; see 6.3.2.1. 70

C++

1.8p1 *The properties of an object are determined when the object is created.*

This referenced/creation difference, compared to C, is possible in C++ because it contains the **`new`** and **`delete`** operators (as language keywords) for dynamic-storage allocation. The type of the object being created is known at the point of creation, which is not the case when the `malloc` library function is used (one of the reasons for the introduction of the concept of effective type in C99).

effective type 948

3.15

3.16

---

recommended practice

recommended practice 72

specification that is strongly recommended as being in keeping with the intent of the standard, but that may be impractical for some implementations

C90

The *Recommended practice* subsections are new in C99.

C++

C++ gives some recommendations inside "[Note: . . .]", but does not explicitly define their status (from reading C++ Committee discussions it would appear to be non-normative).

3.17

---

value

value 73

precise meaning of the contents of an object when interpreted as having a specific type

C++

3.9p4 *The value representation of an object is the set of bits that hold the value of type T.*

3.17.1

---

implementation-defined value

implementation-defined value 74

unspecified value where each implementation documents how the choice is made

C90

Although C90 specifies that implementation-defined values occur in some situations, it never formally defines the term.

C++

The C++ Standard follows C90 in not explicitly defining this term.

3.17.2

75 **indeterminate value** indeterminate value
either an unspecified value or a trap representation

C++

Objects may have an indeterminate value. However, the standard does not explicitly say anything about the properties of this value.

> *. . . , or if the object is uninitialized, a program that necessitates this conversion has undefined behavior.* 4.1p1

3.17.3

3.18

3.19

4. Conformance

82 In this International Standard, "shall" is to be interpreted as a requirement on an implementation or on a program; shall

C++

The C++ Standard does not provide an explicit definition for the term *shall*. However, since the C++ Standard was developed under ISO rules from the beginning, the default ISO rules should apply. ISO shall rules

84 If a "shall" or "shall not" requirement that appears outside of a constraint is violated, the behavior is undefined. shall outside constraint

C++

This specification for the usage of *shall* does not appear in the C++ Standard. The ISO rules specify that the meaning of these terms does not depend on the kind of normative context in which they appear. One implication of this C specification is that the definition of the preprocessor is different in C++. It was essentially copied verbatim from C90, which operated under different *shall* rules :-O. ISO shall rules

85 Undefined behavior is otherwise indicated in this International Standard by the words "undefined behavior" or by the omission of any explicit definition of behavior. undefined behavior indicated by

C++

The C++ Standard does not define the status of any omission of explicit definition of behavior.

88 A program that is correct in all other aspects, operating on correct data, containing unspecified behavior shall be a correct program and act in accordance with 5.1.2.3. correct program

C90

This statement did not appear in the C90 Standard. It was added in C99 to make it clear that a strictly conforming program can contain constructs whose behavior is unspecified, provided the output is not affected by the behavior chosen by an implementation.

C++

1.4p2

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 of this International Standard, a conforming implementation shall, within its resource limits, accept and correctly execute that program.

footnote 3 *"Correct execution" can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.*

Programs which have the status, according to the C Standard, of being *strictly conforming* or *conforming* have no equivalent status in C++.

#error terminate translation

The implementation shall not successfully translate a preprocessing translation unit containing a **#error** preprocessing directive unless it is part of a group skipped by conditional inclusion. 89

C90

C90 required that a diagnostic be issued when a **#error** preprocessing directive was encountered, but the translator was allowed to continue (in the sense that there was no explicit specification saying otherwise) translation of the rest of the source code and signal *successful translation* on completion.

C++

16.5 *. . . , and renders the program ill-formed.*

It is possible that a C++ translator will continue to translate a program after it has encountered a **#error** directive (the situation is as ambiguous as it was in C90).

---

strictly conforming program use features of language/library

A *strictly conforming program* shall use only those features of the language and library specified in this International Standard.[2)] 90

C++

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

The C++ term *well-formed* is not as strong as the C term *strictly conforming*. This is partly as a result of the former language being defined in terms of requirements on an implementation, not in terms of requirements on a program, as in C's case. There is also, perhaps, the thinking behind the C++ term of being able to check statically for a program being well-formed. The concept does not include any execution-time behavior (which strictly conforming does include). The C++ Standard does not define a term stronger than *well-formed*.

standard 1 specifies form and interpretation

The C requirement to use only those library functions specified in the standard is not so clear-cut for freestanding C++ implementations.

1.4p7 *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 (17.4.1.3).*

93 A *conforming hosted implementation* shall accept any strictly conforming program. conforming hosted implementation

C++

No such requirement is explicitly specified in the C++ Standard.

94 A *conforming freestanding implementation* shall accept any strictly conforming program that does not use complex types and in which the use of the features specified in the library clause (clause 7) is confined to the contents of the standard headers **<float.h>**, **<iso646.h>**, **<limits.h>**, **<stdarg.h>**, **<stdbool.h>**, **<stddef.h>**, and **<stdint.h>**. conforming freestanding implementation

C90

The header **<iso646.h>** was added in Amendment 1 to C90. Support for the complex types, the headers **<stdbool.h>** and **<stdint.h>**, are new in C99.

C++

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 include certain language-support libraries (17.4.1.3). 1.4p7

A freestanding implementation has an implementation-defined set of headers. This set shall include at least the following headers, as shown in Table 13: 17.4.1.3p2

...

Table 13 C++ Headers for Freestanding Implementations

Subclause	Header(s)
18.1 Types	**<cstddef>**
18.2 Implementation properties	**<limits>**
18.3 Start and termination	**<cstdlib>**
18.4 Dynamic memory management	**<new>**
18.5 Type identification	**<typeinfo>**
18.6 Exception handling	**<exception>**
18.7 Other runtime support	**<cstdarg>**

The supplied version of the header **<cstdlib>** *shall declare at least the functions* `abort()`, `atexit()`, *and* `exit()` *(18.3).*

The C++ Standard does not include support for the headers **<stdbool.h>** or **<stdint.h>**, which are new in C99.

96 2) A strictly conforming program can use conditional features (such as those in annex F) provided the use is guarded by a **#ifdef** directive with the appropriate macro. footnote 2

C90

The C90 Standard did not contain any conditional constructs.

C++

The C++ Standard also contains optional constructs. However, testing for the availability of any optional constructs involves checking the values of certain class members. For instance, an implementation's support for the IEC 60559 Standard is indicated by the value of the member `is_iec559` (18.2.1.2). 29 IEC 60559

footnote 3

3) This implies that a conforming implementation reserves no identifiers other than those explicitly reserved in this International Standard. 98

C++

The clauses 17.4.3.1, 17.4.4, and their associated subclauses list identifier spellings that are reserved, but do not specify that a conforming C++ implementation must not reserve identifiers having other spellings.

conforming program

A *conforming program* is one that is acceptable to a conforming implementation.4) 99

C++

The C++ conformance model is based on the conformance of the implementation, not a program (1.4p2). However, it does define the term *well-formed program*:

1.3.14 well-formed program

> *a C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).*

implementation document

An implementation shall be accompanied by a document that defines all implementation-defined and locale-specific characteristics and all extensions. 100

C90

locale-specific behavior

Support for locale-specific characteristics is new in C99. The equivalent C90 constructs were defined to be implementation-defined, and hence were also required to be documented.

footnote 4

4) Strictly conforming programs are intended to be maximally portable among conforming implementations. 102

C++

The word *portable* does not occur in the C++ Standard. This may be a consequence of the conformance model which is based on implementations, not programs.

conforming programs may depend on

Conforming programs may depend upon nonportable features of a conforming implementation. 103

C++

While a conforming implementation of C++ may have extensions, 1.4p8, the C++ conformance model does not deal with programs.

5. Environment

environment execution

An implementation translates C source files and executes C programs in two data-processing-system environments, which will be called the *translation environment* and the *execution environment* in this International Standard. 104

C++

The C++ Standard says nothing about the environment in which C++ programs are translated.

Their characteristics define and constrain the results of executing conforming C programs constructed according to the syntactic and semantic rules for conforming implementations. 105

C++

The C++ Standard makes no such observation.

5.1 Conceptual models

5.1.1 Translation environment

5.1.1.1 Program structure

108 The text of the program is kept in units called *source files*, (or *preprocessing files*) in this International Standard. source files preprocessing files

C90

The term *preprocessing files* is new in C99.

C++

The C++ Standard follows the wording in C90 and does not define the term *preprocessing files*.

109 A source file together with all the headers and source files included via the preprocessing directive **#include** is known as a *preprocessing translation unit*. preprocessing translation unit known as

C90

The term *preprocessing translation unit* is new in C99.

C++

Like C90, the C++ Standard does not define the term *preprocessing translation unit*.

110 After preprocessing, a preprocessing translation unit is called a *translation unit*. translation unit known as

C90

> *A source file together with all the headers and source files included via the preprocessing directive* ***#include****, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a translation unit.*

This definition differs from C99 in that it does not specify whether macro definitions are part of a translation unit.

C++

The C++ Standard, 2p1, contains the same wording as C90.

5.1.1.2 Translation phases

115 The precedence among the syntax rules of translation is specified by the following phases.[5)] translation phases of

C++

C++ has nine translation phases. An extra phase has been inserted between what are called phases 7 and 8 in C. This additional phase is needed to handle templates, which are not supported in C. The C++ Standard specifies what the C Rationale calls model A.

116 C++ model A

116 1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary. translation phase 1

C90

In C90 the source file contains characters (the 8-bit kind), not multibyte characters.

C++

2.1p1

> *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. . . . Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character.*

```
#define mkstr(s) #s

char *dollar = mkstr($); // The string "\u0024" is assigned
                          /* The string "$", if that character is supported */
```

C++ model A

Rationale

The C++ Committee defined its Standard in terms of model A, just because that was the clearest to specify (used the fewest hypothetical constructs) because the basic source character set is a well-defined finite set.

The situation is not the same for C given the already existing text for the standard, which allows multibyte characters to appear almost anywhere (the most notable exception being in identifiers), and given the more low-level (or *close to the metal*) nature of some uses of the language.

Therefore, the C committee agreed in general that model B, keeping UCNs and native characters until as late as possible, is more in the “spirit of C” and, while probably more difficult to specify, is more able to encompass the existing diversity. The advantage of model B is also that it might encompass more programs and users’ intents than the two others, particularly if shift states are significant in the source text as is often the case in East Asia.

In any case, translation phase 1 begins with an implementation-defined mapping; and such mapping can choose to implement model A or C (but the implementation must document it). As a by-product, a strictly conforming program cannot rely on the specifics handled differently by the three models: examples of non-strict conformance include handling of shift states inside strings and calls like `fopen("\\ubeda\\file.txt","r")` and `#include "sys\udefault.h"`. Shift states are guaranteed to be handled correctly, however, as long as the implementation performs no mapping at the beginning of phase 1; and the two specific examples given above can be made much more portable by rewriting these as `fopen("\\" "ubeda\\file.txt", "r")` and `#include "sys/udefault.h"`.

---

translation phase 2
physical source line
logical source line

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. 118

C++

The first sentence of 2.1p2 is the same as C90.
The following sentence is not in the C Standard:

2.1p2

> *If, as a result, a character sequence that matches the syntax of a universal-character-name is produced, the behavior is undefined.*

```
#include <stdio.h>

int \u1F\
5F;           // undefined behavior
              /* defined behavior */
void f(void)
```

```
{
printf("\\u0123"); /* No UCNs. */
printf("\\u\
0123"); /* same as above, no UCNs */
        // undefined, character sequence that matches a UCN created
}
```

119 Only the last backslash on any physical source line shall be eligible for being part of such a splice.

C90

This fact was not explicitly specified in the C90 Standard.

C++

The C++ Standard uses the wording from C90.

122 The description is conceptual only, and does not specify any particular implementation.

C++

> *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)* 1.9p1

> *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.* Footnote 5

123 A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place. source file end in new-line

C90

The wording, " . . . before any such splicing takes place.", is new in C99.

129 4. Preprocessing directives are executed, macro invocations are expanded, and **_Pragma** unary operator expressions are executed. translation phase 4

C90

Support for the **_Pragma** unary operator is new in C99.

C++

Support for the **_Pragma** unary operator is new in C99 and is not available in C++.

130 If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined.

C90

Support for universal character names is new in C99.

C++

translation phase 1 116

In C++ universal character names are only processed during translation phase 1. Character sequences created during subsequent phases of translation, which might be interpreted as a universal character name, are not interpreted as such by a translator.

---

preprocessing directives deleted

All preprocessing directives are then deleted. 132

C++

This explicit requirement was added in C99 and is not stated in the C++ Standard.

---

corresponding member if no

if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[7] 134

C90

The C90 Standard did not contain this statement. It was added in C99 to handle the fact that the UCN notation supports the specification of numeric values that may not represent any specified (by ISO 10646) character.

ISO 10646 28

C++

2.2p3 *The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.*

translation phase 1 116

C++ handles implementation-defined character members during translation phase 1.

---

translation phase 6

6. Adjacent string literal tokens are concatenated. 135

C90

6. Adjacent character string literal tokens are concatenated and adjacent wide string literal tokens are concatenated.

It was a constraint violation to concatenate the two types of string literals together in C90. Character and wide string literals are treated on the same footing in C99.

The introduction of the macros for I/O format specifiers in C99 created the potential need to support the concatenation of character string literals with wide string literals. These macros are required to expand to character string literals. A program that wanted to use them in a format specifier, containing wide character string literals, would be unable to do so without this change of specification.

---

translation phase 8

8. All external object and function references are resolved. 139

C++

The C translation phase 8 is numbered as translation phase 9 in C++ (in C++, translation phase 8 specifies the instantiation of templates).

---

footnote 7

7) An implementation need not convert all non-corresponding source characters to the same execution character. 145

C++

The C++ Standard specifies that the conversion is implementation-defined (2.1p1, 2.13.2p5) and does not explicitly specify this special case.

5.1.1.3 Diagnostics

146 A conforming implementation shall produce at least one diagnostic message (identified in an implementation-defined manner) if a preprocessing translation unit or translation unit contains a violation of any syntax rule or constraint, even if the behavior is also explicitly specified as undefined or implementation-defined. diagnostic shall produce

C++

— *If a program contains a violation of any diagnosable rule, a conforming implementation shall issue at least one diagnostic message, except that* 1.4p2

— *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.*

A program that contains "a violation of a rule for which no diagnostic is required", for instance on line 1, followed by "a violation of any diagnosable rule", for instance on line 2; a C++ translator is not required to issue a diagnostic message.

5.1.2 Execution environments

151 All objects with static storage duration shall be *initialized* (set to their initial values) before program startup. static storage duration initialized before startup

C++

In C++ the storage occupied by any object of static storage duration is first zero-initialized at program startup (3.6.2p1, 8.5), before any other initialization takes place. The storage is then initialized by any nonzero values. C++ permits static storage duration objects to be initialized using nonconstant values (not supported in C). The order of initialization is the textual order of the definitions in the source code, within a single translation unit. However, there is no defined order across translation units. Because C requires the values used to initialize objects of static storage duration to be constant, there are no initializer ordering dependencies.

153 *Program termination* returns control to the execution environment.

C++

[Note: in a freestanding environment, start-up and termination is implementation defined; 3.6.1p1

A **`return`** *statement in* `main` *has the effect of leaving the main function (destroying . . . duration) and calling* `exit` *with the return value as the argument.* 3.6.1p5

The function `exit()` *has additional behavior in this International Standard:* 18.3p8

. . . Finally, control is returned to the host environment.

5.1.2.1 Freestanding environment

5.1.2.2 Hosted environment

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

C++

1.4p7 *For a hosted implementation, this International Standard defines the set of available libraries.*

17.4.1.3p1 *For a hosted implementation, this International Standard describes the set of available headers.*

Of course, an implementation is free to produce any number of diagnostics as long as a valid program is still correctly translated. 160

C++

The C++ Standard does not explicitly give this permission. However, producing diagnostic messages that the C++ Standard does not require to be generated might be regarded as an extension, and these are explicitly permitted (1.4p8).

5.1.2.2.1 Program startup

---

or equivalent;[9)] 166

C++

The C++ Standard gives no such explicit permission.

---

or in some other implementation-defined manner. 167

C90

Support for this latitude is new in C99.

C++

The C++ Standard explicitly gives permission for an implementation to define this function using different parameter types, but it specifies that the return type is **`int`**.

3.6.1p2 *It shall have a return type of **`int`**, but otherwise its type is implementation-defined.*

. . .

[Note: it is recommended that any further (optional) parameters be added after `argv`.]

---

main parameters intent

The intent is to supply to the program information determined prior to program startup from elsewhere in the hosted environment. 172

C++

The C++ Standard does not specify any intent behind its support for this functionality.

---

argv lowercase

If the host environment is not capable of supplying strings with letters in both uppercase and lowercase, the implementation shall ensure that the strings are received in lowercase. 173

C++

The C++ Standard is silent on this issue.

177 — The parameters **argc** and **argv** and the strings pointed to by the **argv** array shall be modifiable by the program, and retain their last-stored values between program startup and program termination.

C++

The C++ Standard is silent on this issue.

5.1.2.2.2 Program execution

179 9) Thus, **int** can be replaced by a typedef name defined as **int**, or the type of **argv** can be written as **char ** argv**, and so on. footnote 9

C++

The C++ Standard does not make this observation.

5.1.2.2.3 Program termination

180 If the return type of the **main** function is a type compatible with **int**, a return from the initial call to the **main** function is equivalent to calling the **exit** function with the value returned by the **main** function as its argument;[10)] main return equivalent to

C90

Support for a return type of main other than **int** is new in C99.

C++

The C++ wording is essentially the same as C90.

181 reaching the **}** that terminates the **main** function returns a value of 0.

C90

This requirement is new in C99.

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

182 If the return type is not compatible with **int**, the termination status returned to the host environment is unspecified. main termination status unspecified

C90

Support main returning a type that is not compatible with **int** is new in C99.

C++

> *It shall have a return type of **int**, . . .* 3.6.1p2

Like C90, C++ does not support main having any return type other than **int**.

5.1.2.3 Program execution

189 In the abstract machine, all expressions are evaluated as specified by the semantics. expression evaluation abstract machine

C++

The C++ Standard specifies no such requirement.

signal interrupt abstract machine processing

When the processing of the abstract machine is interrupted by receipt of a signal, only the values of objects as of the previous sequence point may be relied on. 191

C++

1.9p9 *When the processing of the abstract machine is interrupted by receipt of a signal, the value of any objects with type other than* `volatile sig_atomic_t` *are unspecified, and the value of any object not of* `volatile sig_atomic_t` *that is modified by the handler becomes undefined.*

This additional wording closely follows that given in the description of the `signal` function in the library clause of the C Standard.

---

modified objects received correct value

Objects that may be modified between the previous sequence point and the next sequence point need not have received their correct values yet. 192

C++

The C++ Standard does not make this observation.

---

footnote 10

10) In accordance with 6.2.4, the lifetimes of objects with automatic storage duration declared in **`main`** will have ended in the former case, even where they would not have in the latter. 195

C90

This footnote did not appear in the C90 Standard and was added by the response to DR #085.

---

footnote 11

11) The IEC 60559 standard for binary floating-point arithmetic requires certain user-accessible status flags and control modes. 196

C90

The dependence on this floating-point format is new in C99. But, it is still not required.

C++

The C++ Standard does not make these observations about IEC 60559.

Floating-point operations implicitly set the status flags; 197

C++

The C++ Standard does not say anything about status flags in the context of side effects. However, if a C++ implementation supports IEC 60559 (i.e., `is_iec559` is true, 18.2.1.2p52) then floating-point operations will implicitly set the status flags (as required by that standard).

---

modes affect result values of floating-point operations. 198

C++

The C++ Standard does not say anything about floating-point modes in the context of side effects.

---

side effect floating-point state

Implementations that support such floating-point state are required to regard changes to it as side effects—see annex F for details. 199

C++

The C++ Standard does not specify any such requirement.

The floating-point environment library **`<fenv.h>`** provides a programming facility for indicating when these side effects matter, freeing the implementations in other cases. 200

C90

Support for the header **<fenv.h>** is new in C99.

C++

Support for the header **<fenv.h>** is new in C99, and there is no equivalent library header specified in the C++ Standard.

---

201 — At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

C++

> — *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.* 1.9p11

The C++ Standard is technically more accurate in recognizing that the output of a conforming program may vary, if it contains unspecified behavior.

---

209 EXAMPLE 4 Implementations employing wide registers have to take care to honor appropriate semantics. Values are independent of whether they are represented in a register or in memory. For example, an implicit *spilling* of a register is not permitted to alter the value. Also, an explicit *store and load* is required to round to the precision of the storage type. In particular, casts and assignments are required to perform their specified conversion. For the fragment

```
double d1, d2;
float f;
d1 = f = expression;
d2 = (float) expression;
```

the values assigned to **d1** and **d2** are required to have been converted to **float**.

C90

This example is new in C99.

---

210 EXAMPLE 5 Rearrangement for floating-point expressions is often restricted because of limitations in precision as well as range. The implementation cannot generally apply the mathematical associative rules for addition or multiplication, nor the distributive rule, because of roundoff error, even in the absence of overflow and underflow. Likewise, implementations cannot generally replace decimal constants in order to rearrange expressions. In the following fragment, rearrangements suggested by mathematical rules for real numbers are often not valid (see F.8).

```
double x, y, z;
/* ... */
x = (x * y) * z;  // not equivalent to x *= y * z;
z = (x - y) + y ; // not equivalent to z = x;
z = x + x * y;    // not equivalent to z = x * (1.0 + y);
y = x / 5.0;      // not equivalent to y = x * 0.2;
```

C90

This example is new in C99.

---

211 EXAMPLE 6 To illustrate the grouping behavior of expressions, in the following fragment EXAMPLE expression grouping

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

the expression statement behaves exactly the same as

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

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

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

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

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

or

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

since the values for **a** and **b** might have been, respectively, 4 and -8 or -17 and 12. However, on a machine in which overflow silently generates some value and where positive and negative overflows cancel, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur.

C90

The C90 Standard used the term *exception* rather than *trap*.

5.2 Environmental considerations

5.2.1 Character sets

source character set
execution character set

Two sets of characters and their associated collating sequences shall be defined: the set in which source files are written (the *source character set*), and the set interpreted in the execution environment (the *execution character set*). 214

C90

The C90 Standard did not explicitly define the terms *source character set* and *execution character set*.

C++

The C++ Standard does not contain a requirement to define a collating sequence on the character sets it specifies.

basic character set
extended characters

Each set is further divided into a *basic character set*, whose contents are given by this subclause, and a set of zero or more locale-specific members (which are not members of the basic character set) called *extended characters*. 215

C90

source character set 214

This explicit subdivision of characters into sets is new in C99. The wording in the C90 Standard specified the minimum contents of the basic source and basic execution character sets. These terms are now defined exactly, with all other characters being called extended characters.

. . . ; any additional members beyond those required by this subclause are locale-specific.

2.2p3

C++

The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.

The C++ Standard more closely follows the C90 wording.

---

220 it is used to terminate a character string. character string terminate

C++

After any necessary concatenation, in translation phase 7 (2.1), `'\0'` is appended to every string literal so that programs that scan a string can find its end. 2.13.4p4

In practice the C usage is the same as that specified by C++.

---

221 Both the basic source and basic execution character sets shall have the following members: the 26 *uppercase letters* of the Latin alphabet basic source character set basic execution character set

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

the 26 *lowercase letters* of the Latin alphabet

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

the 10 decimal *digits*

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

the following 29 graphic characters

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

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

C90

The C90 Standard referred to these characters as the *English alphabet*.

C++

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 graphics characters: 2.2p1

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

The C++ Standard includes new-line in the basic source character set (C only includes it in the basic execution character set). basic execution character set control characters

The C++ Standard does not separate out the uppercase, lowercase, and decimal digits from the graphical characters, so technically they are not defined for the basic source character set (the library functions such as `toupper` effectively define these terms for the execution character set).

basic character set fit in a byte

The representation of each member of the source and execution basic character sets shall fit in a byte. 222

C++

1.7p1 *A byte is at least large enough to contain any member of the basic execution character set and . . .*

This requirement reverses the dependency given in the C Standard, but the effect is the same.

digit characters contiguous

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. 223

C++

The above wording has been proposed as the response to C++ DR #173.

end-of-line representation

In source files, there shall be some way of indicating the end of each line of text; 224

C++

The C++ Standard does not specify this level of detail (although it does refer to end-of-line indicators, 2.1p1n1).

this International Standard treats such an end-of-line indicator as if it were a single new-line character. 225

C++

2.1p1n1 *. . . (introducing new-line characters for end-of-line indicators) . . .*

If any other characters are encountered in a source file (except in an identifier, a character constant, a string literal, a header name, a comment, or a preprocessing token that is never converted to a token), the behavior is undefined. 227

C90

Support for additional characters in identifiers is new in C99.

C++

2.1p1 *Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character.*

The C++ Standard specifies the behavior and a translator is required to handle source code containing such a character. A C translator is permitted to issue a diagnostic and fail to translate the source code.

letter

A *letter* is an uppercase letter or a lowercase letter as defined above; 228

C90

This definition is new in C99.

in this International Standard the term does not include other characters that are letters in other alphabets. 229

C++

The definition used in the C++ Standard, 17.3.2.1.3 (the footnote applies to C90 only), implies this is also true in C++.

---

230 The universal character name construct provides a way to name other characters.

C90

Support for universal character names is new in C99.

5.2.1.1 Trigraph sequences

5.2.1.2 Multibyte characters

---

238 The source character set may contain multibyte characters, used to represent members of the extended character set. multibyte character source contain

C++

The representations used for multibyte characters, in source code, invariably involve at least one character that is not in the basic source character set:

> *Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character.* 2.1p1

The C++ Standard does not discuss the issue of a translator having to process multibyte characters during translation. However, implementations may choose to replace such characters with a corresponding universal-character-name.

---

239 The execution character set may also contain multibyte characters, which need not have the same encoding as for the source character set.

C++

There is no explicit statement about such behavior being permitted in the C++ Standard. The C header `<wchar.h>` (specified in Amendment 1 to C90) is included by reference and so the support it defines for multibyte characters needs to be provided by C++ implementations.

---

243 — A multibyte character set may have a *state-dependent encoding*, wherein each sequence of multibyte characters begins in an *initial shift state* and enters other locale-specific *shift states* when specific multibyte characters are encountered in the sequence. multibyte character state-dependent encoding shift state

C90

The C90 Standard specified implementation-defined shift states rather than locale-specific shift states.

C++

The definition of multibyte character, 1.3.8, says nothing about encoding issues (other than that more than one byte may be used). The definition of multibyte strings, 17.3.2.1.3.2, requires the multibyte characters to begin and end in the initial shift state.

---

244 While in the initial shift state, all single-byte characters retain their usual interpretation and do not alter the shift state.

C++

The C++ Standard does not explicitly specify this requirement.

---

245 12) The trigraph sequences enable the input of characters that are not defined in the Invariant Code Set as described in ISO/IEC 646, which is a subset of the seven-bit US ASCII code set. footnote 12

C90

The C90 Standard explicitly referred to the 1983 version of ISO/IEC 646 standard.

The interpretation for subsequent bytes in the sequence is a function of the current shift state. 246

C++

A set of virtual functions for handling state-dependent encodings, during program execution, is discussed in Clause 22, Localization library. But, this requirement is not specified.

---

byte all bits zero

— A byte with all bits zero shall be interpreted as a null character independent of shift state. 247

C++

2.2p3 *. . . , plus a null character (respectively, null wide character), whose representation has all zero bits.*

While the C++ Standard does not rule out the possibility of all bits zero having another interpretation in other contexts, other requirements (17.3.2.1.3.1p1 and 17.3.2.1.3.2p1) restrict these other contexts, as do existing character set encodings.

---

multibyte character end in initial shift state

— ~~A byte with all bits zero shall not occur in the second or subsequent bytes of a~~ Such a byte shall not occur as part of any other multibyte character. 248

C++

This requirement can be deduced from the definition of null terminated byte strings, 17.3.2.1.3.1p1, and null terminated multibyte strings, 17.3.2.1.3.2p1.

---

token shift state

— An identifier, comment, string literal, character constant, or header name shall begin and end in the initial shift state. 250

C90

Support for multibyte characters in identifiers is new in C99.

C++

translation phase 1 116

In C++ all characters are mapped to the source character set in translation phase 1. Any shift state encoding will not exist after translation phase 1, so the C requirement is not applicable to C++ source files.

— An identifier, comment, string literal, character constant, or header name shall consist of a sequence of valid multibyte characters. 251

C90

Support for multibyte characters in identifiers is new in C99.

C++

translation phase 1 116

In C++ all characters are mapped to the source character set in translation phase 1. Any shift state encoding will not exist after translation phase 1, so the C requirement is not applicable to C++ source files.

5.2.2 Character display semantics

C++

Clause 18 mentions “display as a wstring” in *Notes:*. But, there is no other mention of display semantics anywhere in the standard.

The *active position* is that location on a display device where the next character output by the `fputc` function would appear. 252

C++

C++ has no concept of active position. The `fputc` function appears in "Table 94" as one of the functions supported by C++.

---

253 The intent of writing a printing character (as defined by the **`isprint`** function) to a display device is to display a graphic representation of that character at the active position and then advance the active position to the next position on the current line.

C++

The C++ Standard does not discuss character display semantics.

---

254 The direction of writing is locale-specific. writing direction locale-specific

C++

The C++ Standard does not discuss character display semantics.

---

255 If the active position is at the final position of a line (if there is one), the behavior of the display device is unspecified.

C++

The C++ Standard does not discuss character display semantics.

---

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

C++

The C++ Standard does not discuss character display semantics.

---

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

C++

Alert appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

> *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:* 17.4.1.2p3

---

258 **`\b`** (*backspace*) Moves the active position to the previous position on the current line. backspace escape sequence

C++

Backspace appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

> *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:* 17.4.1.2p3

---

259 If the active position is at the initial position of a line, the behavior of the display device is unspecified.

C90

If the active position is at the initial position of a line, the behavior is unspecified.

This wording differs from C99 in that it renders the behavior of the program as unspecified. The program simply writes the character; how the device handles the character is beyond its control.

C++

The C++ Standard does not discuss character display semantics.

`\f` (*form feed*) Moves the active position to the initial position at the start of the next logical page. 260

C++

Form feed appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

17.4.1.2p3 *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:*

---

new-line escape sequence

`\n` (*new line*) Moves the active position to the initial position of the next line. 261

C++

New line appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

17.4.1.2p3 *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:*

---

carriage return escape sequence

`\r` (*carriage return*) Moves the active position to the initial position of the current line. 262

C++

Carriage return appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

17.4.1.2p3 *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:*

---

horizontal tab escape sequence

`\t` (*horizontal tab*) Moves the active position to the next horizontal tabulation position on the current line. 263

C++

Horizontal tab appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

17.4.1.2p3 *The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12:*

If the active position is at or past the last defined horizontal tabulation position, the behavior of the display device is unspecified. 264

C90

If the active position is at or past the last defined horizontal tabulation position, the behavior is unspecified.

---

265 `\v` (*vertical tab*) Moves the active position to the initial position of the next vertical tabulation position. vertical tab escape sequence

C++

Vertical tab appears in Table 5, 2.13.2p3. There is no other description of this escape sequence, although the C behavior might be implied from the following wording:

The facilities of the Standard C Library are provided in 18 additional headers, as shown in Table 12: 17.4.1.2p3

---

266 If the active position is at or past the last defined vertical tabulation position, the behavior of the display device is unspecified.

C90

If the active position is at or past the last defined vertical tabulation position, the behavior is unspecified.

---

267 Each of these escape sequences shall produce a unique implementation-defined value which can be stored in a single `char` object. escape sequence fit in char object

C++

This requirement can be deduced from 2.2p3.

---

268 The external representations in a text file need not be identical to the internal representations, and are outside the scope of this International Standard.

C++

The C++ Standard does not get involved in such details.

5.2.3 Signals and interrupts

---

270 **C++**

The C++ Standard specifies, Clause 15 Exception handling, a much richer set of functionality for dealing with exceptional behaviors. While it does not go into the details contained in this C subclause, they are likely, of necessity, to be followed by a C++ implementation.

---

271 Functions shall be implemented such that they may be interrupted at any time by a signal, or may be called by a signal handler, or both, with no alteration to earlier, but still active, invocations' control flow (after the interruption), function return values, or objects with automatic storage duration.

C++

This implementation requirement is not specified in the C++ Standard (1.9p9).

---

272 All such objects shall be maintained outside the *function image* (the instructions that compose the executable representation of a function) on a per-invocation basis. object storage outside function image

C++

The C++ Standard does not contain this requirement.

5.2.4 Environmental limits

environmental limits

Both the translation and execution environments constrain the implementation of language translators and libraries. 273

C++

There is an informative annex which states:

Annex Bp1 *Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process.*

The following summarizes the language-related environmental limits on a conforming implementation; 274

C++

There is an informative annex which states:

Annex Bp2 *The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine conformance.*

the library-related limits are discussed in clause 7. 275

C++

Clause 18.2 contains an *Implementation Limits:*.

5.2.4.1 Translation limits

translation limits

The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:13) 276

C++

Annex Bp2 *However, these quantities are only guidelines and do not determine conformance.*

This wording appears in an informative annex, which itself has no formal status.

limit block nesting

— 127 nesting levels of blocks 277

C90

15 nesting levels of compound statements, iteration control structures, and selection control structures

block 1741 selection statement
block 1742 selection sub-statement
block iteration statement
block loop body

The number of constructs that could create a block increased between C90 and C99, including selection statements and their associated substatements, and iteration statements and their associated bodies. Although use of these constructs doubles the number of blocks created in C99, the limit on the nesting of blocks has increased by a factor of four. So, the conformance status of a program will not be adversely affected.

C++

The following is a non-normative specification.

Nesting levels of compound statements, iteration control structures, and selection control structures [256] Annex Bp2

---

278 — 63 nesting levels of conditional inclusion

C90

8 nesting levels of conditional inclusion

C++

The following is a non-normative specification.

Nesting levels of conditional inclusion [256] Annex Bp2

---

279 — 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration limit type complexity

C++

The following is a non-normative specification.

Pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration [256] Annex Bp2

---

280 — 63 nesting levels of parenthesized declarators within a full declarator limit declarator parentheses

C90

31 nesting levels of parenthesized declarators within a full declarator

C++

The C++ Standard does not discuss declarator parentheses nesting limits.

---

281 — 63 nesting levels of parenthesized expressions within a full expression parenthesized expression nesting levels

C90

31 nesting levels of parenthesized expressions within a full expression

C++

The following is a non-normative specification.

Annex Bp2

Nesting levels of parenthesized expressions within a full expression [256]

---

internal identifier significant characters

— 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character) 282

C90

31 significant initial characters in an internal identifier or a macro name

C++

2.10p1 *All characters are significant.*[20)]

C identifiers that differ after the last significant character will cause a diagnostic to be generated by a C++ translator.

The following is a non-normative specification.

Annex Bp2 *Number of initial characters in an internal identifier or a macro name [1024]*

---

external identifier significant characters

— 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)[14)] 283

C90

6 significant initial characters in an external identifier

C++

2.10p1 *All characters are significant.*[20)]

C identifiers that differ after the last significant character will cause a diagnostic to be generated by a C++ translator.

The following is a non-Normative specification.

Annex Bp2 *Number of initial characters in an external identifier [1024]*

---

limit external identifiers

— 4095 external identifiers in one translation unit 285

C90

511 external identifiers in one translation unit

C++

The following is a non-normative specification.

External identifiers in one translation unit [65536] Annex Bp2

---

286 — 511 identifiers with block scope declared in one block identifiers number in block scope

C90

127 identifiers with block scope declared in one block

C++

The following is a non-normative specification.

Identifiers with block scope declared in one block [1024] Annex Bp2

---

287 — 4095 macro identifiers simultaneously defined in one preprocessing translation unit limit macro definitions

C90

1024 macro identifiers simultaneously defined in one translation unit

C++

The following is a non-normative specification.

Macro identifiers simultaneously defined in one translation unit [65536] Annex Bp2

---

288 — 127 parameters in one function definition limit parameters in definition

C90

31 parameters in one function definition

C++

The following is a non-normative specification.

Annex Bp2

Parameters in one function definition [256]

function call number of arguments

— 127 arguments in one function call 289

C90

31 arguments in one function call

C++

The following is a non-normative specification.

Annex Bp2 *Arguments in one function call [256]*

limit macro parameters

— 127 parameters in one macro definition 290

C90

31 parameters in one macro definition

C++

The following is a non-normative specification.

Annex Bp2 *Parameters in one macro definition [256]*

limit arguments in macro invocation

— 127 arguments in one macro invocation 291

C90

31 arguments in one macro invocation

C++

The following is a non-normative specification.

Annex Bp2 *Arguments in one macro invocation [256]*

limit characters on line

— 4095 characters in a logical source line 292

C90

509 characters in a logical source line

C++

The following is a non-normative specification.

Characters in a logical source line [65536] Annex Bp2

---

293 — 4095 characters in a character string literal or wide string literal (after concatenation) limit string literal

C90

509 characters in a character string literal or wide string literal (after concatenation)

C++

The following is a non-normative specification.

Characters in a character string literal or wide string literal (after concatenation) [65536] Annex Bp2

---

294 — 65535 bytes in an object (in a hosted environment only) limit minimum object size

C90

32767 bytes in an object (in a hosted environment only)

C++

The following is a non-normative specification.

Size of an object [262144] Annex Bp2

---

295 — 15 nesting levels for `#include`d files limit #include nesting

C90

8 nesting levels for #included files

C++

The following is a non-normative specification.

Annex Bp2

Nesting levels for #included files [256]

limit case labels

— 1023 **case** labels for a **switch** statement (excluding those for any nested **switch** statements) 296

C90

*257 case labels for a **switch** statement (excluding those for any nested **switch** statements)*

switch statement 1748 The intent here was to support **switch** statements that included 256 unsigned character values plus EOF (usually implemented as -1).

C++

The following is a non-normative specification.

Annex Bp2 *Case labels for a **switch** statement (excluding those for any nested **switch** statements) [16384]*

limit members in struct/union

— 1023 members in a single structure or union 297

C90

127 members in a single structure or union

C++

The following is a non-normative specification.

Annex Bp2 *Data members in a single class, structure or union [16384]*

limit enumeration constants

— 1023 enumeration constants in a single enumeration 298

C90

127 enumeration constants in a single enumeration

C++

The following is a non-normative specification.

Annex Bp2 *Enumeration constants in a single enumeration [4096]*

limit struct/union nesting

— 63 levels of nested structure or union definitions in a single struct-declaration-list 299

C90

15 levels of nested structure or union definitions in a single struct-declaration-list

C++

The following is a non-normative specification.

Levels of nested class, structure, or union definitions in a single `struct-declaration-list` *[256]* Annex Bp2

5.2.4.2 Numerical limits

300 An implementation is required to document all the limits specified in this subclause, which are specified in the headers **<limits.h>** and **<float.h>**. numerical limits

C90

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

C++

Header **<climits>** *(Table 16): . . . The contents are the same as the Standard C library header* **<limits.h>**. 18.2.2p2

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**. 18.2.2p4

301 Additional limits are specified in **<stdint.h>**.

C90

Support for these limits and the header that contains them is new in C99.

C++

Support for these limits and the header that contains them is new in C99 and is not available in C++.

5.2.4.2.1 Sizes of integer types <limits.h>

303 The values given below shall be replaced by constant expressions suitable for use in **#if** preprocessing directives. integer types sizes

C++

All object-like macros defined by the Standard C library and described in this clause as expanding to integral constant expressions are also suitable for use in **#if** *preprocessing directives, unless explicitly stated otherwise.* 17.4.4.2p2

18.2.2p2

Header **<climits>** *(Table 16): . . . The contents are the same as the Standard C library header* **<limits.h>**

— minimum value for an object of type **long long int** 323

LLONG_MIN -9223372036854775807 // $-(2^{63}-1)$

C90

Support for the type **long long** and its associated macros is new in C99.

C++

The type **long long** is not available in C++ (although many implementations support it).

— maximum value for an object of type **long long int** 324

LLONG_MAX +9223372036854775807 // $2^{63}-1$

C90

Support for the type **long long** and its associated macros is new in C99.

C++

The type **long long** is not available in C++ (although many implementations support it).

— maximum value for an object of type **unsigned long long int** 325

ULLONG_MAX 18446744073709551615 // $2^{64}-1$

C90

Support for the type **unsigned long long** and its associated macros is new in C99.

C++

The type **unsigned long long** is not available in C++.

UCHAR_MAX value

The value **UCHAR_MAX** shall equal $2^{\texttt{CHAR\_BIT}} - 1$. 328

C90

This requirement was not explicitly specified in the C90 Standard.

C++

Like C90, this requirement is not explicitly specified in the C++ Standard.

5.2.4.2.2 Characteristics of floating types <float.h>

floating types characteristics

The characteristics of floating types are defined in terms of a model that describes a representation of floating-point numbers and values that provide information about an implementation's floating-point arithmetic.[16)] 330

C++

18.2.2p4 *Header* **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**

335 p precision (the number of base-b digits in the significand) precision floating-point

C++

The term *significand* is not used in the C++ Standard.

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

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

C90

> *A normalized floating-point number x ($f_1 > 0$ if $x \neq 0$) is defined by the following model:*
>
> $$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, \quad e_{min} \le e \le e_{max}$$

The C90 Standard did not explicitly deal with subnormal or unnormalized floating-point numbers.

C++

The C++ document does not contain any description of a floating-point model. But, Clause 18.2.2 explicitly refers the reader to ISO C90 subclause 5.2.4.2.2

338 In addition to normalized floating-point numbers ($f_1 > 0$ if $x \neq 0$), floating types may be able to contain other kinds of floating-point numbers, such as subnormal floating-point numbers ($x \neq 0$, $e = e_{min}$, $f_1 = 0$) and unnormalized floating-point numbers ($x \neq 0$, $e > e_{min}$, $f_1 = 0$), and values that are not floating-point numbers, such as infinities and NaNs. floating types can represent

C90

The C90 Standard does not mention these kinds of floating-point numbers. However, the execution environments for C90 programs are likely to be the same as C99 in terms of their support for IEC 60559.

C++

The C++ Standard does not go into this level of detail.

339 A *NaN* is an encoding signifying Not-a-Number. NaN

C90

This concept was not described in the C90 Standard.

C++

Although this concept was not described in C90, C++ does include the concept of NaN.

18.2.1.2p34 Template class numeric_limits

```
static const bool has_quiet_NaN;
```

> *True if the type has a representation for a quiet (non-signaling) "Not a Number."*[193)]

18.2.2.1p37

```
static const bool has_signaling_NaN;
```

> *True if the type has a representation for a signaling "Not a Number."*[194)]

NaN raising an exception

A *quiet NaN* propagates through almost every arithmetic operation without raising a floating-point exception; 340

C90

The concept of NaN was not discussed in the C90 Standard.

C++

18.2.1.2p34

```
static const bool has_quiet_NaN;
```

True if the type has a representation for a quiet (non-signaling) "Not a Number."[193)]

---

a *signaling NaN* generally raises a floating-point exception when occurring as an arithmetic operand.[17)] 341

C++

18.2.1.2p37

```
static const bool has_signaling_NaN;
```

True if the type has a representation for a signaling "Not a Number."[194]

---

footnote 16

16) The floating-point model is intended to clarify the description of each floating-point characteristic and does not require the floating-point arithmetic of the implementation to be identical. 345

C++

The C++ Standard does not explicitly describe a floating-point model. However, it does include the template class `numeric_limits`. This provides a mechanism for accessing the values of many, but not all, of the characteristics used by the C model to describe its floating-point model.

---

floating-point operations accuracy

The accuracy of the floating-point operations (`+`, `-`, `*`, `/`) and of the library functions in **`<math.h>`** and **`<complex.h>`** that return floating-point results is implementation-defined , as is the accuracy of the conversion between floating-point internal representations and string representations performed by the library routine in **`<stdio.h>`**, **`<stdlib.h>`** and **`<wchar.h>`**. 346

C90

In response to DR #063 the Committee stated (while the Committee did revisit this issue during the C99 revision of the C Standard, there was no change of requirements):

DR #063

Probably the most useful response would be to amend the C Standard by adding two requirements on implementations:

Require that an implementation document the maximum errors it permits in arithmetic operations and in evaluating math functions. These should be expressed in terms of "units in the least-significant position" (ULP) or "lost bits of precision."

Establish an upper bound for these errors that all implementations must adhere to. The state of the art, as the Committee understands it, is:

correctly rounded results for arithmetic operations (no loss of precision)

1 ULP for functions such as sqrt, sin, and cos (loss of 1 bit of precision)

4–6 ULP (loss of 2–3 bits of precision) for other math functions.

Since not all commercially viable machines and implementations meet these exacting requirements, the C Standard should be somewhat more liberal.

*The Committee would, however, suggest a requirement no more liberal than a loss of 3 bits of precision, out of kindness to users. An implementation with worse performance can always conform by providing a more conservative version of **<float.h>**, even if that is not a desirable approach in the general case. The Committee should revisit this issue during the revision of the C Standard.*

C++

The C++ Standard says nothing on this issue.

---

347 The implementation may state that the accuracy is unknown.

C++

The C++ Standard does not explicitly give this permission.

---

348 All integer values in the **<float.h>** header, except **FLT_ROUNDS**, shall be constant expressions suitable for use in **#if** preprocessing directives; float.h suitable for #if

C90

*Of the values in the **<float.h>** header, FLT_RADIX shall be a constant expression suitable for use in **#if** preprocessing directives;*

C99 requires a larger number of values to be constant expressions suitable for use in a **#if** preprocessing directive and in static and aggregate initializers.

C++

The requirement in C++ only applies if the header **<cfloat>** is used (17.4.1.2p3). While this requirement does not apply to the contents of the header **<float.h>**, it is very likely that implementations will meet it and no difference is flagged here. The namespace issues associated with using **<cfloat>** do not apply to names defined as macros in C (17.4.1.2p4)

*All object-like macros defined by the Standard C library and described in this clause as expanding to integral constant expressions are also suitable for use in **#if** preprocessing directives, unless explicitly stated otherwise.* 17.4.4.2p2

The C++ wording does not specify the C99 Standard and some implementations may only support the requirements specified in C90.

---

349 all floating values shall be constant expressions.

C90

all other values need not be constant expressions.

This specification has become more restrictive, from the implementations point of view, in C99.

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

All except **DECIMAL_DIG**, **FLT_EVAL_METHOD**, **FLT_RADIX**, and **FLT_ROUNDS** have separate names for all three floating-point types. 350

C90

Support for DECIMAL_DIG and FLT_EVAL_METHOD is new in C99. The FLT_EVAL_METHOD macro appears to add functionality that could cause a change of behavior in existing programs. However, in practice it provides access to information on an implementation's behavior that was not previously available at the source code level. Implementations are not likely to change their behavior because of this macro, other than to support it.

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

The floating-point model representation is provided for all values except **FLT_EVAL_METHOD** and **FLT_ROUNDS**. 351

C90

Support for FLT_EVAL_METHOD is new in C99.

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

FLT_ROUNDS

The rounding mode for floating-point addition is characterized by the implementation-defined value of **FLT_ROUNDS**:[18)] 352

-1 indeterminable
0 toward zero
1 to nearest
2 toward positive infinity
3 toward negative infinity

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

C++

It is possible that some implementations will only meet the requirements contained in the C90 Standard.

The C++ header **<limits>** also contains the enumerated type:

18.2.1.3

```
namespace std {
   enum float_round_style {
     round_indeterminable       = -1,
     round_toward_zero          = 0,
     round_to_nearest           = 1,
     round_toward_infinity      = 2,
     round_toward_neg_infinity  = 3
   };
}
```

which is referenced by the following member, which exists for every specialization of an arithmetic type (in theory this allows every floating-point type to support a different rounding mode):

18.2.1.2p62

Meaningful for all floating point types.

```
static const float_round_style round_style;
```

The rounding style for the type.[206)]

353 ~~The~~ Except for assignment and cast (which remove all extra range and precision), the values of operations with floating operands and values subject to the usual arithmetic conversions and of floating constants are evaluated to a format whose range and precision may be greater than required by the type. floating operands evaluation format

C90

> *The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type;* 6.2.1.5

This wording allows wider representations to be used for floating-point operands and expressions. It could also be interpreted (somewhat liberally) to support the idea that C90 permitted floating constants to be represented in wider formats where the usual arithmetic conversions applied.
Having the representation of floating constants change depending on how an implementation chooses to specify FLT_EVAL_METHOD is new in C99.

C++

Like C90, the FLT_EVAL_METHOD macro is not available in C++.

354 The use of evaluation formats is characterized by the implementation-defined value of **FLT_EVAL_METHOD**:[19) FLT_EVAL_METHOD

C90

Support for the FLT_EVAL_METHOD macro is new in C99. Its significant attendant baggage was also present in C90 implementations, but was explicitly not highlighted in that standard.

C++

Support for the FLT_EVAL_METHOD macro is new in C99 and it is not available in C++. However, it is likely that the implementation of floating point in C++ will be the same as in C.

358 For implementations that do not support IEC 60559:1989, the terms quiet NaN and signaling NaN are intended to apply to encodings with similar behavior.

C90

The concept of NaN is new, in terms of being explicitly discussed, in C99.

C++

18.2.1.2p34

```
static const bool has_quiet_NaN;
```

> *True if the type has a representation for a quiet (non-signaling) "Not a Number."*[193)
>
> *Meaningful for all floating point types.*
>
> *Shall be true for all specializations in which* `is_iec559 != false`.

18.2.1.2p37

```
static const bool has_signaling_NaN;
```

> *True if the type has a representation for a signaling "Not a Number." 194)*
>
> *Meaningful for all floating point types.*
>
> *Shall be true for all specializations in which is_iec559 != false.*

18.2.1.2p52

```
static const bool is_iec559;
```

True if and only if the type adheres to IEC 559 standard.[201)]

The C++ Standard requires NaNs to be supported if IEC 60559 is supported, but says nothing about the situation where that standard is not supported by an implementation.

---

footnote 18 18) Evaluation of **FLT_ROUNDS** correctly reflects any execution-time change of rounding mode through the function **fesetround** in <**fenv.h**>. 359

C90

Support for the header <**fenv.h**> is new in C99. The C90 Standard did not provide a mechanism for changing the rounding direction.

C++

Support for the header <**fenv.h**> and the `fesetround` function is new in C99 and is not specified in the C++ Standard.

---

footnote 19 19) The evaluation method determines evaluation formats of expressions involving all floating types, not just real types. 360

C90

Support for complex types is new in C99.

C++

The complex types are a template class in C++. The definitions of the instantiation of these classes do not specify that the evaluation format shall be the same as for the real types. But then, the C++ Standard does not specify the evaluation format for the real types.

For example, if **FLT_EVAL_METHOD** is 1, then the product of two **float _Complex** operands is represented in the **double _Complex** format, and its parts are evaluated to **double**. 361

C++

The C++ Standard does not specify a `FLT_EVAL_METHOD` mechanism.

The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater or equal in magnitude (absolute value) to those shown, with the same sign: 365

C90

In C90 the only expression that was required to be a constant expression was `FLT_RADIX`. It was explicitly stated that the others need not be constant expressions; however, in most implementations, the values were constant expressions.

C++

18.2.1p3

For all members declared `static const` in the `numeric_limits` template, specializations shall define these values in such a way that they are usable as integral constant expressions.

366 — radix of exponent representation, b FLT_RADIX

FLT_RADIX **2**

C++

18.2.1.2p16

```
static const int radix;
```

For floating types, specifies the base or radix of the exponent representation (often 2).[185]

368 — number of decimal digits, n, such that any floating-point number in the widest supported floating type with p_{max} radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value, DECIMAL_DIG macro

$$\begin{cases} p_{max} \log_{10} b & \text{if } b \text{ is a power of 10} \\ \lceil 1 + p_{max} \log_{10} b \rceil & \text{otherwise} \end{cases}$$

DECIMAL_DIG **10**

C90

Support for the DECIMAL_DIG macro is new in C99.

C++

Support for the DECIMAL_DIG macro is new in C99 and specified in the C++ Standard.

369 — number of decimal digits, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix b digits and back again without change to the q decimal digits, *_DIG macros

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

FLT_DIG **6**
DBL_DIG **10**
LDBL_DIG **10**

C++

18.2.1.2p9

```
static const int digits10;
```

Number of base 10 digits that can be represented without change.

Footnote 184

Equivalent to FLT_DIG, DBL_DIG, LDBL_DIG.

18.2.2p4

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

---

*_MIN_EXP — minimum negative integer such that **FLT_RADIX** raised to one less than that power is a normalized floating-point number, e_{min} 370

FLT_MIN_EXP
DBL_MIN_EXP
LDBL_MIN_EXP

C++

18.2.1.2p23

```
static const int min_exponent;
```

Minimum negative integer such that `radix` *raised to that power is in the range of normalised floating point numbers.*[189)]

Footnote 189 *Equivalent to FLT_MIN_EXP, DBL_MIN_EXP, LDBL_MIN_EXP.*

18.2.2p4 *Header* **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

---

*_MIN_10_EXP — minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers, $\lceil \log_{10} b^{e_{min}-1} \rceil$ 371

FLT_MIN_10_EXP	**-37**
DBL_MIN_10_EXP	**-37**
LDBL_MIN_10_EXP	**-37**

C++

18.2.1.2p25

```
static const int min_exponent10;
```

Minimum negative integer such that 10 raised to that power is in the range of normalised floating point numbers.[190)]

Footnote 190 *Equivalent to FLT_MIN_10_EXP, DBL_MIN_10_EXP, LDBL_MIN_10_EXP.*

18.2.2p4

*Header **<cfloat>** (Table 17): . . . The contents are the same as the Standard C library header **<float.h>**.*

---

372 — maximum integer such that **FLT_RADIX** raised to one less than that power is a representable finite floating-point number, e_{max} *_MAX_EXP

FLT_MAX_EXP
DBL_MAX_EXP
LDBL_MAX_EXP

C++

18.2.1.2p27

```
static const int max_exponent;
```

Maximum positive integer such that `radix` raised to the power one less than that integer is a representable finite floating point number.[191)]

Footnote 191

Equivalent to FLT_MAX_EXP, DBL_MAX_EXP, LDBL_MAX_EXP.

18.2.2p4

*Header **<cfloat>** (Table 17): . . . The contents are the same as the Standard C library header **<float.h>**.*

---

373 — maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers, $\lfloor \log_{10}((1 - b^{-p})b^{e_{max}}) \rfloor$ *_MAX_10_EXP

FLT_MAX_10_EXP	**+37**
DBL_MAX_10_EXP	**+37**
LDBL_MAX_10_EXP	**+37**

C++

18.2.1.2p29

```
static const int max_exponent10;
```

Maximum positive integer such that 10 raised to that power is in the range of normalised floating point numbers.

Footnote 192

Equivalent to FLT_MAX_10_EXP, DBL_MAX_10_EXP, LDBL_MAX_10_EXP.

18.2.2

Header **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

---

floating values listed

The values given in the following list shall be replaced by constant expressions with implementation-defined values that are greater than or equal to those shown: 374

C90

C90 did not contain the requirement that the values be constant expressions.

C++

This requirement is not specified in the C++ Standard, which refers to the C90 Standard by reference.

— maximum representable finite floating-point number, $(1 - b^{-p})b^{e_{max}}$ 375

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

C++

18.2.1.2p4

```
static T max() throw();
```

Maximum finite value.[182]

Footnote 182 *Equivalent to CHAR_MAX, SHRT_MAX, FLT_MAX, DBL_MAX, etc.*

18.2.2p4 *Header* **<cfloat>** *(Table 17): . . . The contents are the same as the Standard C library header* **<float.h>**.

---

*_EPSILON

— the difference between 1 and the least value greater than 1 that is representable in the given floating point type, b^{1-p} 377

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

C++

18.2.1.2p20

```
static T epsilon() throw();
```

Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.[187)]

Footnote 187

Equivalent to FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON.

*Header **<cfloat>** (Table 17): . . . The contents are the same as the Standard C library header **<float.h>**.* 18.2.2p4

---

378 — minimum normalized positive floating-point number, $b^{e_{min}-1}$ *_MIN macros

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

C++

18.2.1.2p1

```
static T min() throw();
```

Maximum finite value.[181]

Equivalent to CHAR_MIN, SHRT_MIN, FLT_MIN, DBL_MIN, etc. Footnote 181

*Header **<cfloat>** (Table 17): . . . The contents are the same as the Standard C library header **<float.h>**.* 18.2.2p4

Recommended practice

---

379 Conversion from (at least) **double** to decimal with **DECIMAL_DIG** digits and back should be the identity function. DECIMAL_DIG conversion recommended practice

C90

The *Recommended practice* clauses are new in the C99 Standard.

C++

There is no such macro, or requirement specified in the C++ Standard.

---

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

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

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

FLT_RADIX	**2**
DECIMAL_DIG	**17**
FLT_MANT_DIG	**24**

```
FLT_EPSILON       1.19209290E-07F // decimal constant
FLT_EPSILON             0X1P-23F // hex constant
FLT_DIG                        6
FLT_MIN_EXP                 -125
FLT_MIN           1.17549435E-38F // decimal constant
FLT_MIN                0X1P-126F // hex constant

FLT_MIN_10_EXP               -37
FLT_MAX_EXP                 +128
FLT_MAX           3.40282347E+38F // decimal constant
FLT_MAX           0X1.fffffeP127F // hex constant
FLT_MAX_10_EXP               +38
DBL_MANT_DIG                  53
DBL_EPSILON 2.2204460492503131E-16 // decimal constant
DBL_EPSILON              0X1P-52 // hex constant
DBL_DIG                       15
DBL_MIN_EXP                -1021
DBL_MIN    2.2250738585072014E-308 // decimal constant
DBL_MIN                0X1P-1022 // hex constant
DBL_MIN_10_EXP              -307
DBL_MAX_EXP                +1024
DBL_MAX    1.7976931348623157E+308 // decimal constant
DBL_MAX     0X1.ffffffffffffP1023 // hex constant
DBL_MAX_10_EXP              +308
```

If a type wider than **double** were supported, then **DECIMAL_DIG** would be greater than 17. For example, if the widest type were to use the minimal-width IEC 60559 double-extended format (64 bits of precision), then **DECIMAL_DIG** would be 21.

C90

The C90 wording referred to the ANSI/IEEE-754–1985 standard.

6. Language

6.1 Notation

In the syntax notation used in this clause, syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. 384

C++

1.6p1 *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.*

The C++ grammar contains significantly more syntactic ambiguities than C. Some implementations have used mainly syntactic approaches to resolving these,[6] while others make use of semantic information to guide the parse.[1] For instance, knowing what an identifier has been declared as, simplifies the parsing of the following:

```
template_name     < a , b > - 5  // equivalent to (template_name < a , b >) - 5)
non_template_name < a , b > - 5  // equivalent to (non_template_name < a) , (b > - 5)
```

---

385 A colon (:) following a nonterminal introduces its definition.

C++

The C++ Standard does not go into this level of detail (although it does use this notation).

---

388 When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

C90

This convention was not explicitly specified in the C90 Standard.

C++

The C++ Standard does not explicitly specify the conventions used. However, based on the examples given in clause 1.6 and usage within the standard, the conventions used appear to be the reverse of those used in C (i.e., syntactic categories are italicized and words are separated by hyphens).

---

389 A summary of the language syntax is given in annex A.

C90

The summary appeared in Annex B of the C90 Standard, and this fact was not pointed out in the normative text.

6.2 Concepts

6.2.1 Scopes of identifiers

---

392 a tag or a member of a structure, union, or enumeration;

C++

The C++ Standard does not define the term *tag*. It uses the terms *enum-name* (7.2p1) for enumeration definitions and *class-name* (9p1) for classes.

---

396 or a macro parameter. identifier macro parameter

C++

The C++ Standard does not list macro parameters as one of the entities that can be denoted by an identifier.

---

397 The same identifier can denote different entities at different points in the program. identifier denote different entities

C++

The C++ Standard does not explicitly state this possibility, although it does include wording (e.g., 3.3p4) that implies it is possible.

---

398 A member of an enumeration is called an *enumeration constant*. enumeration constant

C++

There is no such explicit definition in the C++ Standard (7.2p1 comes close), although the term *enumeration constant* is used.

visible identifier scope

For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. 400

C++

3.3p1 *In general, each particular name is valid only within some possibly discontiguous portion of program text called its scope.*

3.3p4 *Local extern declarations (3.5) may introduce a name into the declarative region where the declaration appears and also introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to both regions.*

3.3.7p5 *If a name is in scope and is not hidden it is said to be visible.*

same identifier

Different entities designated by the same identifier either have different scopes, or are in different name spaces. 401

C90

In all but one case, duplicate label names having the same identifier designate different entities in the same scope, or in the same name space, was a constraint violation in C90. Having the same identifier denote two different labels in the same function caused undefined behavior. The wording in C99 changed to make this case a constraint violation.

label name unique 1725

C++

The C++ Standard does not explicitly make this observation, although it does include wording (e.g., 3.6.1p3) that implies it is possible.

scope kinds of

There are four kinds of scopes: function, file, block, and function prototype. 402

C++

The C++ Standard does not list the possible scopes in a single sentence. There are subclauses of 3.3 that discuss the five kinds of C++ scope: function, namespace, local, function prototype, and class. A C declaration at file scope is said to have *namespace scope* or *global scope* in C++. A C declaration with block scope is said to have *local scope* in C++. Class scope is what appears inside the curly braces in a structure/union declaration (or other types of declaration in C++).

Given the following declaration, at file scope:

```
struct S {
        int m; /* has file scope */
               // has class scope
        } v;   /* has file scope */
               // has namespace scope
```

file scope

If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit. 407

C++

> *A name declared outside all named or unnamed namespaces (7.3), blocks (6.3), function declarations (8.3.5), function definitions (8.4) and classes (9) has global namespace scope (also called global scope). The potential scope of such a name begins at its point of declaration (3.3.1) and ends at the end of the translation unit that is its declarative region.* 3.3.5p3

---

408 If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block. block scope terminates

C++

> *A name declared in a block (6.3) is local to that block. Its potential scope begins at its point of declaration (3.3.1) and ends at the end of its declarative region.* 3.3.2p1

> *The potential scope of a function parameter name in a function definition (8.4) begins at its point of declaration. . . . , else it ends at the outermost block of the function definition.* 3.3.2p2

---

410 If an identifier designates two different entities in the same name space, the scopes might overlap. scope overlapping

C90

This sentence does not appear in the C90 Standard, but the situation it describes could have occurred in C90.

C++

> *The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name.* 3.3p1

---

411 If so, the scope of one entity (the *inner scope*) will be a strict subset of the scope of the other entity (the *outer scope*). scope inner scope outer

C++

The C observation can be inferred from the C++ wording.

> *In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.* 3.3p1

---

413 the entity declared in the outer scope is *hidden* (and not visible) within the inner scope. outer scope identifier hidden

C++

The C rules are a subset of those for C++ (3.3p1), which include other constructs. For instance, the scope resolution operator, `::`, allows a file scope identifier to be accessed, but it does not introduce that identifier into the current scope.

Unless explicitly stated otherwise, where this International Standard uses the term “identifier” to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs. 414

C90

There is no such statement in the C90 Standard.

C++

There is no such statement in the C++ Standard (which does contain uses of *identifier* that refer to its syntactic form).

---

scope same

Two identifiers have the *same scope* if and only if their scopes terminate at the same point. 415

C90

Although the wording of this sentence is the same in C90 and C99, there are more blocks available to have their scopes terminated in C99. The issues caused by this difference are discussed in the relevant sentences for iteration-statement, loop body, a *`selection-statement`* a substatement associated with a selection statement.

block iteration statement
block loop body
block 1741 selection statement
block 1742 selection sub-statement

C++

The C++ Standard uses the term *same scope* (in the sense “in the same scope”, but does not provide a definition for it. Possible interpretations include using the common English usage of the word *same* or interpreting the following wording

3.3p1

> *In general, each particular name is valid only within some possibly discontiguous portion of program text called its scope. To determine the scope of a declaration, it is sometimes convenient to refer to the potential scope of a declaration. The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name. In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.*

to imply that the scope of the first declaration of a is not the same as the scope of b in the following:

```
{
int a;
int b;
   {
   int a;
   }
}
```

---

tag scope begins

Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag. 416

C++

The C++ Standard defines the term *point of declaration* (3.3.1p1). The C++ point of declaration of the identifier that C refers to as a tag is the same (3.3.1p5). The scope of this identifier starts at the same place in C and C++ (3.3.2p1, 3.3.5p3).

417 Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. enumeration constant scope begins

C++

> *The point of declaration for an enumerator is immediately after its* `enumerator-definition`. *[Example:* 3.3.1p3
>
> ```
> const int x = 12;
>
> { enum { x = x }; }
> ```
>
> *Here, the enumerator* `x` *is initialized with the value of the constant* `x`*, namely 12.]*

In C, the first declaration of x is not a constant expression. Replacing it by a definition of an enumeration of the same name would have an equivalent, conforming effect in C.

418 Any other identifier has scope that begins just after the completion of its declarator. identifier scope begins

C++

The C++ Standard defines the potential scope of an identifier having either local (3.3.2p1) or global (3.3.5p3) scope to begin at its *point of declaration* (3.3.1p1). However, there is no such specification for identifiers having function prototype scope (which means that in the following declaration the second occurrence of `p1` might not be considered to be in scope).

```
void f(int p1, int p2@lsquare[]sizeof(p1)@rsquare[]);
```

No difference is flagged here because it is not thought likely that C++ implementation will behave different from C implementations in this case.

6.2.2 Linkages of identifiers

420 An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*.[21)] linkage

C++

The C++ Standard also defines the term *linkage*. However, it is much less relaxed about multiple declarations of the same identifier (3.3p4).

> *A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:* 3.5p2

421 There are three kinds of linkage: external, internal, and none. linkage kinds of

C++

The C++ Standard defines the three kinds of linkage: external, internal, and no linkage. However, it also defines the concept of *language linkage*:

> *All function types, function names, and variable names have a language linkage. [Note: Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here. For example, a particular language linkage may be associated with a particular form of representing names of* 7.5p1

objects and functions with external linkage, or with a particular calling convention, etc.] The default language linkage of all function types, function names, and variable names is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.

---

object external linkage denotes same function external linkage denotes same

In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with *external linkage* denotes the same object or function. 422

C++

The situation in C++ is complicated by its explicit support for linkage to identifiers whose definition occurs in other languages and its support for overloaded functions (which is based on a function's signature (1.3.10) rather than its name). As the following references show, the C++ Standard does not appear to explicitly specify the same requirements as C.

3.2p3 *Every program shall contain exactly one definition of every non-inline function or object that is used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 12.1, 12.4 and 12.8). An inline function shall be defined in every translation unit in which it is used.*

C++ 1350 one definition rule

Some of the consequences of the C++ *one definition rule* are discussed elsewhere.

3.5p2 *A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:*

— When a name has external linkage, the entity it denotes can be referred to by names from scopes of other translation units or from other scopes of the same translation unit.

7.5p6 *At most one function with a particular name can have C language linkage.*

---

no linkage identifier declaration is unique

Each declaration of an identifier with *no linkage* denotes a unique entity. 424

C++

The C++ *one definition rule* covers most cases:

3.2p1 *No translation unit shall contain more than one definition of any variable, function, class type, enumeration type or template.*

However, there is an exception:

7.1.3p2 *In a given scope, a **typedef** specifier can be used to redefine the name of any type declared in that scope to refer to the type to which it already refers. [Example:*

```
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;
```

—end example]

Source developed using a C++ translator may contain duplicate typedef names that will generate a constraint violation if processed by a C translator.

The following does not prohibit names from the same scope denoting the same entity:

> *A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:* 3.5p2
>
> *— When a name has no linkage, the entity it denotes cannot be referred to by names from other scopes.*

This issue is also discussed elsewhere. 1350 declaration only one if no linkage

---

425 If the declaration of a file scope identifier for an object or a function contains the storage-class specifier **`static`**, the identifier has internal linkage.[22)] static internal linkage

C++

> *A name having namespace scope (3.3.5) has internal linkage if it is the name of* 3.5p3
>
> *— an object, reference, function or function template that is explicitly declared **`static`** or,*
>
> *— an object or reference that is explicitly declared **`const`** and neither explicitly declared **`extern`** nor previously declared to have external linkage; or*

```
const int glob; /* external linkage */
                // internal linkage
```

Adhering to the guideline recommendations dealing with textually locating declarations in a header file and including these headers, ensures that this difference in behavior does not occur (or will at least cause a diagnostic to be generated if they do). ?? identifier declared in one file ?? identifier definition shall #include

---

426 For an identifier declared with the storage-class specifier **`extern`** in a scope in which a prior declaration of that identifier is visible,[23)] if the prior declaration specifies internal or external linkage, the linkage of the identifier at the later declaration is the same as the linkage specified at the prior declaration. extern identifier linkage same as prior declaration

C90

The wording in the C90 Standard was changed to its current form by the response to DR #011.

---

427 21) There is no linkage between different identifiers. footnote 21

C++

The C++ Standard says this the other way around.

> *A name is said to have linkage when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:* 3.5p2

---

428 22) A function declaration can contain the storage-class specifier **`static`** only if it is at file scope; see 6.7.1. footnote 22

C++

7.1.1p4

There can be no **`static`** *function declarations within a block, . . .*

This wording does not require a diagnostic, but the exact status of a program containing such a usage is not clear.

A function can be declared as a static member of a **`class`** (**`struct`**). Such usage is specific to C++ and cannot occur in C.

---

prior declaration not

If no prior declaration is visible, or if the prior declaration specifies no linkage, then the identifier has external linkage. 429

C90

The wording in the C90 Standard was changed to its current form by the response to DR #011.

---

member no linkage

an identifier declared to be anything other than an object or a function; 433

C++

3.5p3 *A name having namespace scope (3.3.5) has internal linkage if it is the name of*

— a data member of an anonymous union.

While the C Standard does not support anonymous unions, some implementations support it as an extension.

3.5p4 *A name having namespace scope (3.3.5) has external linkage if it is the name of*

— a named class (clause 9), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (7.1.3); or

— a named enumeration (7.2), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (7.1.3); or

— an enumerator belonging to an enumeration with external linkage; or

3.5p8 *Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage.*

The following C definition may cause a link-time failure in C++. The names of the enumeration constants are not externally visible in C, but they are in C++. For instance, the identifiers `E1` or `E2` may be defined as externally visible objects or functions in a header that is not included by the source file containing this declaration.

```
extern enum T {E1, E2} glob;
```

There are also some C++ constructs that have no meaning in C, or would be constraint violations.

```
void f()
{
union {int a; char *p; }; /* not an object */
                          // an anonymous union object

/*
 * The following all have meaning in C++
```

```
 *
   a=1;
 *
   p="Derek";
 */
}
```

---

435 a block scope identifier for an object declared without the storage-class specifier **extern**. no linkage block scope object

C++

> *Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage. A name with no linkage (notably, the name of a class or enumeration declared in a local scope (3.3.2)) shall not be used to declare an entity with linkage.* 3.5p8

The following conforming C function is ill-formed in C++.

```
void f(void)
{
typedef int INT;

extern INT a; /* Strictly conforming */
              // Ill-formed

enum E_TAG {E1, E2};

extern enum E_TAG b; /* Strictly conforming */
                     // Ill-formed
}
```

---

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

C++

The C++ Standard does not specify that the behavior is undefined and gives an example (3.5p6) showing that the behavior is defined.

6.2.3 Name spaces of identifiers

---

438 If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. name space

C++

This C statement is not always true in C++, where the name lookup rules can involve semantics as well as syntax; for instance, in some cases the **struct** can be omitted. 441 tag name space

---

439 Thus, there are separate *name spaces* for various categories of identifiers, as follows:

C++

C++ uses **namespace** as a keyword (there are 13 syntax rules associated with it and an associated keyword, **using**) and as such it is denotes a different concept from the C name space. C++ does contain some of the name space concepts present in C, and even uses the term namespace to describe them (which can be somewhat confusing). These are dealt with under the relevant sentences that follow.

label name space

— *label names* (disambiguated by the syntax of the label declaration and use); 440

C++

6.1p1 *Labels have their own name space and do not interfere with other identifiers.*

tag name space

— the *tags* of structures, unions, and enumerations (disambiguated by following any[24] of the keywords **struct**, **union**, or **enum**); 441

C++

Tags in C++ exist in what is sometimes known as *one and a half name spaces*. Like C they can follow the keywords **struct**, **union**, or **enum**. Under certain conditions, the C++ Standard allows these keywords to be omitted.

3.4p1 *The name lookup rules apply uniformly to all names (including `typedef-names` (7.1.3), `namespace-names` (7.3) and `class-names` (9.1)) wherever the grammar allows such names in the context discussed by a particular rule.*

In the following:

```
struct T {int i;};
struct S {int i;};
int T;

void f(T p); // Ill-formed, T is an int object
             /* Constraint violation */

void g(S p); // Well-formed, C++ allows the struct keyword to be omitted
             // There is only one S visible at this point
             /* Constraint violation */
```

C source code migrated to C++ will contain the **struct/union** keyword. C++ source code being migrated to C, which omits the *`class-key`*, will cause a diagnostic to be generated.

The C++ rules for tags and typedefs sharing the same identifier are different from C.

3.4.4p2 *If the name in the `elaborated-type-specifier` is a simple identifier, and unless the `elaborated-type-specifier` has the following form:*

`class-key identifier ;`

the identifier is looked up according to 3.4.1 but ignoring any non-type names that have been declared. If this name lookup finds a `typedef-name`, the `elaborated-type-specifier` is ill-formed.

The following illustrates how a conforming C and C++ program can generate different results:

```
extern int T;

int size(void)
{
struct T {
        double mem;
        };

return sizeof(T);   /* sizeof(int) */
                    // sizeof(struct T)
}
```

The following example illustrates a case where conforming C source is ill-formed C++.

```
struct TAG {int i;};
typedef float TAG;

struct TAG x; /* does not affect the conformance status of the program */
              // Ill-formed
```

442 — the *members* of structures or unions; members name space

C++

> *The following rules describe the scope of names declared in classes.* 3.3.6p1

In C++ members exist in a scope, not a name space.

```
struct {
       enum E_TAG { E1, E2}  /* C identifiers have file scope */
                             // C++ identifiers have class scope
                             m1;
       } x;

enum E_TAG y; /* C conforming */
              // C++ no identifier names E_TAG is visible here
```

443 each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the `.` or `->` operator); member namespace

C++

> *The name of a class member shall only be used as follows:* 3.3.6p2
>
> . . .
>
> — *after the* `.` *operator applied to an expression of the type of its class (5.2.5) or a class derived from its class,*
>
> — *after the* `->` *operator applied to a pointer to an object of its class (5.2.5) or a class derived from its class,*

```
struct {
       enum {E1, E2} m;
       } x;

x.m = E1; /* does not affect the conformance status of the program */
          // ill-formed. X::E1 is conforming C++ but a syntax violation in C
```

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

C++

The C++ Standard does not define the term *ordinary identifiers*, or another term similar to it.

footnote 24

24) There is only one name space for tags even though three are possible. 447

C++

There is no separate name space for tags in C++. They exist in the same name space as object/function/typedef *ordinary identifiers*.

6.2.4 Storage durations of objects

storage duration object

An object has a *storage duration* that determines its lifetime. 448

C++

1.8p1 *An object has a storage duration (3.7) which influences its lifetime (3.8).*

In C++ the initialization and destruction of many objects is handled automatically and in an undefined order (exceptions can alter the lifetime of an object, compared to how it might appear in the visible source code). For these reasons an object's storage duration does not fully determine its lifetime, it only influences it.

There are three storage durations: static, automatic, and allocated. 449

C90

The term *allocated* storage duration did not appear in the C90 Standard. It was added by the response to DR #138.

C++

3.7p1 *The storage duration is determined by the construct used to create the object and is one of the following:*

— *static storage duration*

— *automatic storage duration*

— *dynamic storage duration*

The C++ term *dynamic storage* is commonly used to describe the term *allocated storage*, which was introduced in C99.

lifetime of object

The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. 451

C90

The term *lifetime* was used twice in the C90 Standard, but was not defined by it.

C++

3.8p1 *The lifetime of an object is a runtime property of the object. The lifetime of an object of type* `T` *begins when:*

. . .

The lifetime of an object of type `T` *ends when:*

The following implies that storage is allocated for an object during its lifetime:

3.7p1

> *Storage duration is the property of an object that defines the minimum potential lifetime of the storage containing the object.*

---

452 An object exists, has a constant address,[25] and retains its last-stored value throughout its lifetime.[26]

C++

There is no requirement specified in the C++ Standard for an object to have a constant address. The requirements that are specified include:

> *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).* 1.9p10

> *All objects which neither have dynamic storage duration nor are local have static storage duration. The storage for these objects shall last for the duration of the program (3.6.2, 3.6.3).* 3.7.1p1

---

453 If an object is referred to outside of its lifetime, the behavior is undefined.

C++

The C++ Standard does not unconditionally specify that the behavior is undefined (the cases associated with pointers are discussed in the following C sentence):

> *The properties ascribed to objects throughout this International Standard apply for a given object only during its lifetime. [Note: in particular, before the lifetime of an object starts and after its lifetime ends there are significant restrictions on the use of the object, as described below, in 12.6.2 and in 12.7. describe the behavior of objects during the construction and destruction phases.]* 3.8p3

---

454 The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime. pointer indeterminate

C++

The C++ Standard is less restrictive; if does not specify that the value of the pointer becomes indeterminate.

> *Before the lifetime of an object has started but after the storage which the object will occupy has been allocated[34] or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. Such a pointer refers to allocated storage (3.7.3.2), and using the pointer as if the pointer were of type `void*`, is well-defined. Such a pointer may be dereferenced but the resulting lvalue may only be used in limited ways, as described below. If the object will be or was of a class type with a nontrivial destructor, and the pointer is used as the operand of a `delete-expression`, the program has undefined behavior.* 3.8p5

Source developed using a C++ translator may contain pointer accesses that will cause undefined behavior when a program image created by a C implementation is executed.

---

455 An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **`static`** has *static storage duration*. static storage duration

C++

The wording in the C++ Standard is not based on linkage and corresponds in many ways to how C developers often deduce the storage duration of objects.

3.7.1p1 *All objects which neither have dynamic storage duration nor are local have static storage duration.*

3.7.1p3 *The keyword* **`static`** *can be used to declare a local variable with static storage duration.*

---

automatic storage duration

An object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*. 457

C++

3.7.2p1 *Local objects explicitly declared* **`auto`** *or* **`register`** *or not explicitly declared* **`static`** *or* **`extern`** *have automatic storage duration.*

---

object lifetime from entry to exit of block

For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. 458

C++

3.7.2p1 *The storage for these objects lasts until the block in which they are created exits.*

5.2.2p4 *The lifetime of a parameter ends when the function in which it is defined returns.*

6.7p2 *Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).*

Which is a different way of phrasing 3.7.2p1.

3.8p1 *The lifetime of an object of type* `T` *begins when:*

— storage with the proper alignment and size for type `T` *is obtained, and*

. . .

The lifetime of an object of type `T` *ends when:*

— the storage which the object occupies is reused or released.

The C++ Standard does not appear to completely specify when the lifetime of objects created on entry into a block begins.

459 (Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.)

C++

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). 1.9p10

The storage for these objects lasts until the block in which they are created exits. 3.7.2p1

460 If the block is entered recursively, a new instance of the object is created each time. block entered recursively

C90

The C90 Standard did not point this fact out.

C++

As pointed out elsewhere, the C++ Standard does not explicitly specify when storage for such objects is created. However, recursive instances of block scope declarations are supported. 458 object lifetime from entry to exit of block

Recursive calls are permitted, except to the function named `main` *(3.6.1).* 5.2.2p9

Variables with automatic storage duration (3.7.2) are initialized each time their `declaration-statement` *is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).* 6.7p2

461 The initial value of the object is indeterminate. object initial value indeterminate

C++

If no initializer is specified for an object, and the object is of (possibly cv-qualified) non-POD class type (or array thereof), the object shall be default-initialized; if the object is of const-qualified type, the underlying class type shall have a user-declared default constructor. 8.5p9

Otherwise, if no initializer is specified for an object, the object and its subobjects, if any, have an indeterminate initial value[90]; if the object or any of its subobjects are of const-qualified type, the program is ill-formed.

C does not have constructors. So a const-qualified object definition, with a structure or union type, would be ill-formed in C++.

```
struct T {int i;};

void f(void)
{
const struct T loc; /* very suspicious, but conforming */
                    // Ill-formed
}
```

initialization performed every time declaration reached

If an initialization is specified for the object, it is performed each time the declaration is reached in the execution of the block; 462

C90

> *If an initialization is specified for the value stored in the object, it is performed on each normal entry, but not if the block is entered by a jump to a labeled statement.*

Support for mixing statements and declarations is new in C99. The change in wording is designed to ensure that the semantics of existing C90 programs is unchanged by this enhanced functionality.

VLA lifetime starts/ends

For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[27] 464

C90

Support for variable length arrays is new in C99.

C++

C++ does not support variable length arrays in the C99 sense; however, it does support containers:

23.1p1
> *Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.*

The techniques involved, templates, are completely different from any available in C and are not discussed further here.

---

footnote 25

25) The term “constant address” means that two pointers to the object constructed at possibly different times will compare equal. 468

C++

The C++ Standard is silent on this issue.

The address may be different during two different executions of the same program. 469

C++

The C++ Standard does not go into this level of detail.

---

footnote 26

26) In the case of a volatile object, the last store need not be explicit in the program. 470

C++

7.1.5.1p8
> *[Note:* **`volatile`** *is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. See 1.9 for detailed semantics. In general, the semantics of* **`volatile`** *are intended to be the same in C++ as they are in C.]*

---

footnote 27

27) Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration. 471

C90

Support for VLAs is new in C99.

6.2.5 Types

475 Types are partitioned into *object types* (types that fully describe objects), *function types* (types that describe functions), and *incomplete types* (types that describe objects but lack information needed to determine their sizes). types partitioned object types incomplete types

C++

> *Incompletely-defined object types and the void types are incomplete types (3.9.1).* 3.9p6

So in C++ the set of incomplete types and the set of object types overlap each other.

> *An object type is a (possibly cv-qualified) type that is not a function type, not a reference type, and not a **`void`** type.* 3.9p9

A sentence in the C++ Standard may be word-for-word identical to one appearing in the C Standard and yet its meaning could be different if the term *object type* is used. The aim of these C++ subclauses is to point out such sentences where they occur.

476 An object declared as type `_Bool` is large enough to store the values 0 and 1. _Bool large enough to store 0 and 1

C90

Support for the type `_Bool` is new in C99.

C++

> *Values of type **`bool`** are either **`true`** or **`false`**.* 3.9.1p6

> *An rvalue of type **`bool`** can be converted to an rvalue of type **`int`**, with **`false`** becoming zero and **`true`** becoming one.* 4.5p4

> *A **`bool`** value can successfully be stored in a bit-field of any nonzero size.* 9.6p3

477 An object declared as type `char` is large enough to store any member of the basic execution character set. char hold any member of execution character set

C++

> *Objects declared as characters (**`char`**) shall be large enough to store any member of the implementation's basic character set.* 3.9.1p1

The C++ Standard does not use the term character in the same way as C. 59 character single-byte

basic character set positive if stored in char object

If a member of the basic execution character set is stored in a **char** its value is guaranteed to be ~~positive~~non-negative. 478

C++

The following is really a tautology since a single character literal is defined to have the type **char** in C++.

3.9.1p1 *If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character.*

The C++ does place a requirement on the basic execution character set used.

2.2p3 *For each basic execution character set, the values of the members shall be non-negative and distinct from one another.*

If any other character is stored in a **char** object, the resulting value is implementation-defined but shall be within the range of values that can be represented in that type. 479

C90

*If other quantities are stored in a **char** object, the behavior is implementation-defined: the values are treated as either signed or nonnegative integers.*

The implementation-defined behavior referred to in C90 was whether the values are treated as signed or nonnegative, not the behavior of the store. This C90 wording was needed as part of the chain of deduction that the plain char type behaved like either the signed or unsigned character types. This requirement was made explicit in C99. In some cases the C90 behavior for storing *other characters* in a **char** object could have been undefined (implicitly). The effect of the change to the C99 behavior is at most to turn undefined behavior into implementation-defined behavior. As such, it does not affect conforming programs.

char 516 range, representation and behavior

Issues relating to this sentence were addressed in the response to DR #040, question 7.

C++

2.2p3 *The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.*

glyph

The only way of storing a particular character (using a glyph typed into the source code) into a char object is through a character constant, or a string literal.

2.13.2p1 *An ordinary character literal that contains a single `c-char` has type **char**, with value equal to the numerical value of the encoding of the `c-char` in the execution character set.*

2.13.4p1 *An ordinary string literal has type "array of n `const char`" and static storage duration (3.7), where n is the size of the string as defined below, and is initialized with the given characters.*

3.9.1p1

If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character.

Taken together these requirements are equivalent to those given in the C Standard.

---

480 There are five *standard signed integer types*, designated as **signed char**, **short int**, **int**, **long int**, and **long long int**. standard signed integer types

C90

Support for the type **long long int** (and its unsigned partner) is new in C99.

C++

Support for the type **long long int** (and its unsigned partner) is new in C99 and is not available in C++. (It was discussed, but not adopted by the C++ Committee.) Many hosted implementations support these types.

---

482 There may also be implementation-defined *extended signed integer types*.[28)] extended signed integer types

C90

The C90 Standard never explicitly stated this, but it did allow extensions. However, the response to DR #067 specified that the types of `size_t` and `ptrdiff_t` must be selected from the list of integer types specified in the standard. An extended type cannot be used.

C++

The C++ Standard does not explicitly specify an implementation's ability to add extended signed integer types, but it does explicitly allow extensions (1.4p8).

---

483 The standard and extended signed integer types are collectively called *signed integer types*.[29)] signed integer types

C90

Explicitly including the extended signed integer types in this definition is new in C99.

---

487 The type **_Bool** and the unsigned integer types that correspond to the standard signed integer types are the *standard unsigned integer types*. standard unsigned integer

C90

Support for the type **_Bool** is new in C99.

C++

In C++ the type **bool** is classified as an integer type (3.9.1p6). However, it is a type distinct from the signed and unsigned integer types.

---

489 The standard and extended unsigned integer types are collectively called *unsigned integer types*.[30)] unsigned integer types

C90

Explicitly including the extended signed integer types in the definition is new in C99.

---

490 28) Implementation-defined keywords shall have the form of an identifier reserved for any use as described in 7.1.3. footnote 28

C90

The C90 Standard did not go into this level of detail on implementation extensions.

C++

The C++ Standard does not say anything about how an implementation might go about adding additional keywords. However, it does list a set of names that is always reserved for the implementation (17.4.3.1.2).

footnote 29

29) Therefore, any statement in this Standard about signed integer types also applies to the extended signed integer types. 491

C++

The C++ Standard does not place any requirement on extended integer types that may be provided by an implementation.

---

standard integer types
extended integer types

The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*, the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*. 493

C++

3.9.1p7

> *Types **bool**, **char**, **wchar_t**, and the signed and unsigned integer types are collectively called integral types.*[43)] *A synonym for integral type is integer type.*

Footnote 43

> *43) Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to **int**, **unsigned int**, **long**, or **unsigned long**, as specified in 4.5.*

enumeration constant type 864

The issue of enumerations being distinct types, rather than integer types, is discussed elsewhere.

---

integer types relative ranges
rank relative ranges

For any two integer types with the same signedness and different integer conversion rank (see 6.3.1.1), the range of values of the type with smaller integer conversion rank is a subrange of the values of the other type. 494

C90

There was no requirement in C90 that the values representable in an unsigned integer type be a subrange of the values representable in unsigned integer types of greater rank. For instance, a C90 implementation could make the following choices:

```
SHRT_MAX  ==   32767  /* 15 bits */
USHRT_MAX ==  262143  /* 18 bits */
INT_MAX   ==   65535  /* 16 bits */
UINT_MAX  ==  131071  /* 17 bits */
```

No C90 implementation known to your author fails to meet the stricter C99 requirement.

C++

3.9.1p2

> *In this list, each type provides at least as much storage as those preceding it in the list.*

C++ appears to have a lower-level view than C on integer types, defining them in terms of storage allocated, rather than the values they can represent. Some deduction is needed to show that the C requirement on values also holds true for C++:

3.9.1p3

> *For each of the signed integer types, there exists a corresponding (but different) unsigned integer type: "**unsigned char**", "**unsigned short int**", "**unsigned int**", and "**unsigned long int**," each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type*[40)] *;*

They occupy the same storage,

> *that is, each signed integer type has the same object representation as its corresponding unsigned integer type. The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.* 3.9.1p3

they have a common set of values, and

> *Unsigned integers, declared* **`unsigned`***, shall obey the laws of arithmetic modulo* 2^n *where* n *is the number of bits in the value representation of that particular size of integer.*[41)] 3.9.1p4

more values can be represented as the amount of storage increases.

QED.

---

497 There are three *real floating types*, designated as **`float`**, **`double`**, and **`long double`**.[32)] floating types three real

C90

What the C90 Standard calls *floating types* includes complex types in C99. The term *real floating types* is new in C99.

---

500 There are three *complex types*, designated as **`float _Complex`**, **`double _Complex`**, and **`long double _Complex`**.[33)] complex types

C90

Support for complex types is new in C99.

C++

In C++ `complex` is a template class. Three specializations are defined, corresponding to each of the floating-point types.

> *The header* **`<complex>`** *defines a template class, and numerous functions for representing and manipulating complex numbers.* 26.2p1

> *The effect of instantiating the template* `complex` *for any type other than float, double, or long double is unspecified.* 26.2p2

The C++ syntax for declaring objects of type **`complex`** is different from C.

```
#ifdef __cplusplus

#include <complex>

typedef complex<float> float_complex;
typedef complex<double> double_complex;
typedef complex<long double> long_double_complex;

#else

#include <complex.h>

typedef float complex float_complex;
typedef double complex double_complex;
typedef long double complex long_double_complex;
#endif
```

floating types

The real floating and complex types are collectively called the *floating types*. 501

C90

What the C90 Standard calls *floating types* includes complex types in C99.

For complex types, it is the type given by deleting the keyword **`_Complex`** from the type name. 504

C++

In C++ the complex type is a template class and declarations involving it also include a floating-point type bracketed between `< >` tokens. This is the type referred to in the C99 wording.

---

complex type representation

Each complex type has the same representation and alignment requirements as an array type containing exactly two elements of the corresponding real type; 505

C++

The C++ Standard defines `complex` as a template class. There are no requirements on an implementation's representation of the underlying values.

---

complex component representation

the first element is equal to the real part, and the second element to the imaginary part, of the complex number. 506

C++

Clause 26.2.3 lists the first parameter of the complex constructor as the real part and the second parameter as the imaginary part. But, this does not imply anything about the internal representation used by an implementation.

---

basic types

The type **`char`**, the signed and unsigned integer types, and the floating types are collectively called the *basic types*. 507

C++

The C++ Standard uses the term *basic types* three times, but never defines it:

3.9.1p10

> *[Note: even if the implementation defines two or more basic types to have the same value representation, they are nevertheless different types.]*

13.5p7

> *The identities among certain predefined operators applied to basic types (for example, `++a` ≡ `a+=1`) need not hold for operator functions. Some predefined operators, such as `+=`, require an operand to be an lvalue when applied to basic types; this is not required by operator functions.*

Footnote 174

> *174) An implicit exception to this rule are types described as synonyms for basic integral types, such as `size_t` (18.1) and `streamoff` (27.4.1).*

---

footnote 33

33) A specification for imaginary types is in informative annex G. 511

C++

There is no such annex in the C++ Standard.

512 34) An implementation may define new keywords that provide alternative ways to designate a basic (or any other) type; footnote 34

C90

Defining new keywords that provide alternative ways of designating basic types was not discussed in the C90 Standard.

C++

The object-oriented constructs supported by C++ removes most of the need for implementations to use additional keywords to designate basic (or any other) types

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

C++

Clause 3.9.1p1 does not explicitly define the term *character types*, but the wording implies the same definition as C.

516 The implementation shall define **char** to have the same range, representation, and behavior as either **signed char** or **unsigned char**.[35] char range, representation and behavior

C90

This sentence did not appear in the C90 Standard. Its intent had to be implied from wording elsewhere in that standard.

C++

> *A **char**, a **signed char**, and an **unsigned char** occupy the same amount of storage and have the same alignment requirements (3.9); that is, they have the same object representation.* 3.9.1p1
>
> . . .
>
> *In any particular implementation, a plain **char** object can take on either the same values as **signed char** or an **unsigned char**; which one is implementation-defined.*

In C++ the type **char** can cause different behavior than if either of the types **signed char** or **unsigned char** were used. For instance, an overloaded function might be defined to take each of the three distinct character types. The type of the argument in an invocation will then control which function is invoked. This is not an issue for C code being translated by a C++ translator, because it will not contain overloaded functions.

518 Each distinct enumeration constitutes a different *enumerated type*. enumeration different type

C++

The C++ Standard also contains this sentence (3.9.2p1). But it does not contain the integer compatibility requirements that C contains. The consequences of this are discussed elsewhere. 1447 enumeration type compatible with

519 The type **char**, the signed and unsigned integer types, and the enumerated types are collectively called *integer types*. integer types

C90

In the C90 Standard these types were called either *integral types* or *integer types*. DR #067 lead to these two terms being rationalized to a single term.

C++

3.9.1p7 *Types **bool**, **char**, **wchar_t**, and the signed and unsigned integer types are collectively called integral types.*[43)] *A synonym for integral type is integer type.*

In C the type `_Bool` is an unsigned integer type and `wchar_t` is compatible with some integer type. In C++ they are distinct types (in overload resolution a **bool** or **wchar_t** will not match against their implementation-defined integer type, but against any definition that uses these named types in its parameter list).

In C++ the enumerated types are not integer types; they are a compound type, although they may be converted to some integer type in some contexts.

standard integer types 493

---

real types

The integer and real floating types are collectively called *real types*. 520

C90

C90 did not include support for complex types and this definition is new in C99.

C++

The C++ Standard follows the C90 Standard in its definition of integer and floating types.

---

arithmetic type

Integer and floating types are collectively called *arithmetic types*. 521

C90

Exactly the same wording appeared in the C90 Standard. Its meaning has changed in C99 because the introduction of complex types has changed the definition of the term floating types.

floating types three real 497

C++

The wording in 3.9.1p8 is similar (although the C++ complex type is not a basic type).

The meaning is different for the same reason given for C90.

---

type domain

Each arithmetic type belongs to one *type domain*: the *real type domain* comprises the real types, the *complex type domain* comprises the complex types. 522

C90

Support for complex types and the concept of type domain is new in C99.

C++

In C++ `complex` is a class defined in one of the standard headers. It is treated like any other class. There is no concept of type domain in the C++ Standard.

---

void is incomplete type

The **void** type comprises an empty set of values; 523

C90

base document

The **void** type was introduced by the C90 Committee. It was not defined by the base document.

---

derived type

Any number of *derived types* can be constructed from the object, function, and incomplete types, as follows: 525

C++

C++ has derived classes, but it does not define derived types as such. The term *compound types* fills a similar role:

3.9.2p1 *Compound types can be constructed in the following ways:*

527 Array types are characterized by their element type and by the number of elements in the array.

C++

The two uses of the word *characterized* in the C++ Standard do not apply to array types. There is no other similar term applied to array types (8.3.4) in the C++ Standard.

528 An array type is said to be derived from its element type, and if its element type is *T*, the array type is sometimes called "array of *T*".

C++

This usage of the term *derived from* is not applied to types in C++; only to classes. The C++ Standard does not define the term *array of T*. However, the usage implies this meaning and there is also the reference:

> *("array of unknown bound of* `T`*" and "array of N* `T`*")* 3.9p7

529 The construction of an array type from an element type is called "array type derivation".

C++

This kind of terminology is not defined in the C++ Standard.

530 — A *structure type* describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type. structure type sequentially allocated objects

C90

Support for a member having an incomplete array type is new in C99.

C++

C++ does not have structure types, it has class types. The keywords **`struct`** and **`class`** may both be used to define a class (and *plain old data* structure and union types). The keyword **`struct`** is supported by C++ for backwards compatibility with C.

> *Nonstatic data members of a (non-union) class declared without an intervening* `access-specifier` *are allocated so that later members have higher addresses within a class object.* 9.2p12

C does not support static data members in a structure, or access-specifiers.

> *— classes containing a sequence of objects of various types (clause 9), a set of types, enumerations and functions for manipulating these objects (9.3), and a set of restrictions on the access to these entities (clause 11);* 3.9.2p1

Support for a member having an incomplete array type is new in C99 and not is supported in C++.

> *In such cases, and except for the declaration of an unnamed bit-field (9.6), the* `decl-specifier-seq` *shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration.* 7p3

The only members that can have their names omitted in C are bit-fields. Thus, taken together the above covers the requirements specified in the C90 Standard.

531 — A *union type* describes an overlapping nonempty set of member objects, each of which has an optionally specified name and possibly distinct type. union type overlapping members

C++

9.5p1 *Each data member is allocated as if it were the sole member of a struct.*

This implies that the members overlap in storage.

3.9.2p1 *— unions, which are classes capable of containing objects of different types at different times, 9.5;*

7p3 *In such cases, and except for the declaration of an unnamed bit-field (9.6), the* `decl-specifier-seq` *shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration.*

The only members that can have their names omitted in C are bit-fields. Thus, taken together the preceding covers the requirements specified in the C Standard.

---

function type

— A *function type* describes a function with specified return type. 532

C++

3.9.2p1 *— functions, which have parameters of given types and return* **`void`** *or references or objects of a given type, 8.3.5;*

The parameters, in C++, need to be part of a function's type because they may be needed for overload resolution. This difference is not significant to developers using C because it does not support overloaded functions.

A function type is characterized by its return type and the number and types of its parameters. 533

C++

C++ defines and uses the concept of function *signature* (1.3.10), which represents information amount the number and type of a function's parameters (not its return type). The two occurrences of the word *characterizes* in the C++ Standard are not related to functions.

---

function returning T

A function type is said to be derived from its return type, and if its return type is *T*, the function type is sometimes called "function returning *T*". 534

C++

The term *function returning T* appears in the C++ Standard in several places; however, it is never formally defined.

The construction of a function type from a return type is called "function type derivation". 535

C++

There is no such definition in the C++ Standard.

---

footnote 35

35) **`CHAR_MIN`**, defined in **`<limits.h>`**, will have one of the values 0 or **`SCHAR_MIN`**, and this can be used to distinguish the two options. 536

C++

The C++ Standard includes the C90 library by reference. By implication, the preceding is also true in C++.

537 Irrespective of the choice made, **char** is a separate type from the other two and is not compatible with either. char separate type

C++

> *Plain **char**, **signed char**, and **unsigned char** are three distinct types.* 3.9.1p1

538 36) Since object types do not include incomplete types, an array of incomplete type cannot be constructed. footnote 36

C++

> *Incompletely-defined object types and the void types are incomplete types (3.9.1).* 3.9p6

The C++ Standard makes a distinction between incompletely-defined object types and the **void** type. 475 object types

> *The declared type of an array object might be an array of incomplete class type and therefore incomplete; if the class type is completed later on in the translation unit, the array type becomes complete; the array type at those two points is the same type.* 3.9p7

The following deals with the case where the size of an array may be omitted in a declaration:

> *When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions that specify the bounds of the arrays can be omitted only for the first member of the sequence.* 8.3.4p3

Arrays of incomplete structure and union types are permitted in C++.

```
{
struct st;
typedef struct st_0 A@lsquare[]4@rsquare[]; /* Undefined behavior */
                          // May be well- or ill-formed
typedef struct st_1 B@lsquare[]4@rsquare[]; /* Undefined behavior */
                          // May be well- or ill-formed
struct st_0 {             /* nothing has changed */
             int mem;     // declaration of A becomes well-formed
            };
}                         /* nothing has changed */
                          // declaration of B is now known to be ill-formed
```

539 — A *pointer type* may be derived from a function type, an object type, or an incomplete type, called the *referenced type*. pointer type referenced type

C++

C++ includes support for what it calls *reference* types (8.3.2), so it is unlikely to use the term *referenced type* in this context (it occurs twice in the standard). There are requirements in the C++ Standard (5.3.1p1) that apply to pointers to object and function types, but there is no explicit discussion of how they might be created.

542 The construction of a pointer type from a referenced type is called "pointer type derivation".

C++

The C++ Standard does not define this term, although the term `derived-declarator-type-list` is defined (8.3.1p1).

---

scalar types

Arithmetic types and pointer types are collectively called *scalar types*. 544

C++

3.9p10 *Arithmetic types (3.9.1), enumeration types, pointer types, and pointer to member types (3.9.2), and* `cv-qualified` *versions of these types (3.9.3) are collectively called scalar types.*

While C++ includes type qualifier in the definition of scalar types, this difference in terminology has no impact on the interpretation of constructs common to both languages.

---

aggregate type

Array and structure types are collectively called *aggregate types*.[37)] 545

C++

8.5.1p1 *An aggregate is an array or a class (clause 9) with no user-declared constructors (12.1), no private or protected non-static data members (clause 11), no base classes (clause 10), and no virtual functions (10.3).*

Class types in C++ include union types. The C definition of aggregate does not include union types. The difference is not important because everywhere that the C++ Standard uses the term *aggregate* the C Standard specifies aggregate and union types.

The list of exclusions covers constructs that are in C++, but not C. (It does not include static data members, but they do not occur in C and are ignored during initialization in C++.) There is one place in the C++ Standard (3.10p15) where the wording suggests that the C definition of aggregate is intended.

---

array type completed by

It is completed, for an identifier of that type, by specifying the size in a later declaration (with internal or external linkage). 547

C++

3.9p7 *The declared type of an array object might be an array of unknown size and therefore be incomplete at one point in a translation unit and complete later on;*

Which does not tell us how it got completed. Later on in the paragraph we are given the example:

3.9p7

```
extern int arr@lsquare[]@rsquare[];   // the type of arr is incomplete

int arr@lsquare[]10@rsquare[];        // now the type of arr is complete
```

which suggests that an array can be completed, in a later declaration, by specifying that it has 10 elements. :-)

---

incomplete type completed by

It is completed, for all declarations of that type, by declaring the same structure or union tag with its defining content later in the same scope. 550

C++

3.9p7

A class type (such as "`class X`") might be incomplete at one point in a translation unit and complete later on;

An example later in the same paragraph says:

```
class X;              // X is an incomplete type
struct X { int i; };  // now X is a complete type
```
3.9p7

The following specifies when a class type is completed; however, it does not list any scope requirements.

A class is considered a completely-defined object type (3.9) (or complete type) at the closing `}` of the `class-specifier`. 9.2p2

In practice the likelihood of C++ differing from C, in scope requirements on the completion of types, is small and no difference is listed here.

---

551 Array, function, and pointer types are collectively called *derived declarator types*. derived declarator types

C++

There is no equivalent term defined in the C++ Standard.

---

552 A *declarator type derivation* from a type *T* is the construction of a derived declarator type from *T* by the application of an array-type, a function-type, or a pointer-type derivation to *T*.

C++

There is no equivalent definition in the C++ Standard, although the description of compound types (3.9.2) provides a superset of this definition.

---

553 A type is characterized by its *type category*, which is either the outermost derivation of a derived type (as noted above in the construction of derived types), or the type itself if the type consists of no derived types. type category

C++

The term *outermost level* occurs in a few places in the C++ Standard, but the term *type category* is not defined.

---

554 Any type so far mentioned is an *unqualified type*. unqualified type

C++

In C++ it is possible for the term *type* to mean a qualified or an unqualified type (3.9.3).

---

555 Each unqualified type has several *qualified versions* of its type,[38] corresponding to the combinations of one, two, or all three of the **`const`**, **`volatile`**, and **`restrict`** qualifiers. qualified type versions of

C90

The **`noalias`** qualifier was introduced in later drafts of what was to become C90. However, it was controversial and there was insufficient time available to the Committee to resolve the issues involved. The noalias qualifier was removed from the document, prior to final publication. The **`restrict`** qualifier has the same objectives as **`noalias`**, but specifies the details in a different way.

Support for the **`restrict`** qualifier is new in C99.

C++

3.9.3p1

Each type which is a cv-unqualified complete or incomplete object type or is **`void`** *(3.9) has three corresponding cv-qualified versions of its type: a* `const-qualified` *version, a* `volatile-qualified` *version, and a* `const-volatile-qualified` *version.*

The **`restrict`** qualifier was added to C99 while the C++ Standard was being finalized. Support for this keyword is not available in C++.

---

alignment pointer to structures
representation pointer to structures

All pointers to structure types shall have the same representation and alignment requirements as each other. 560

C90

This requirement was not explicitly specified in the C90 Standard.

C++

The C++ Standard follows the C90 Standard in not explicitly stating any such requirement.

---

alignment pointer to unions
representation pointer to unions

All pointers to union types shall have the same representation and alignment requirements as each other. 561

C90

This requirement was not explicitly specified in the C90 Standard.

C++

The C++ Standard follows the C90 Standard in not explicitly stating any such requirement.

---

alignment pointers

Pointers to other types need not have the same representation or alignment requirements. 562

C++

3.9.2p3 *The value representation of pointer types is implementation-defined.*

---

footnote 37

37) Note that aggregate type does not include union type because an object with union type can only contain one member at a time. 563

C++

The C++ Standard does include union types within the definition of aggregate types, 8.5.1p1. So, this rationale was not thought applicable by the C++ Committee.

6.2.6 Representations of types

6.2.6.1 General

---

types representation

The representations of all types are unspecified except as stated in this subclause. 569

C90

This subclause is new in C99, although some of the specifications it contains were also in the C90 Standard.

C++

3.9p1 *[Note: 3.9 and the subclauses thereof impose requirements on implementations regarding the representation of types.*

These C++ subclauses go into some of the details specified in this C subclause.

570 Except for bit-fields, objects are composed of contiguous sequences of one or more bytes, the number, order, and encoding of which are either explicitly specified or implementation-defined. object contiguous sequence of bytes

C++

> *An object of POD[4] type (3.9) shall occupy contiguous bytes of storage.* 1.8p5

The acronym POD stands for *Plain Old Data* and is intended as a reference to the simple, C model, or laying out objects. A POD type is any scalar type and some, C compatible, structure and union types.

In general the C++ Standard says nothing about the number, order, or encoding of the bytes making up what C calls an object, although C++ does specify the same requirements as C on the layout of members of a structure or union type that is considered to be a POD.

571 Values stored in unsigned bit-fields and objects of type `unsigned char` shall be represented using a pure binary notation.[40] unsigned char pure binary

C90

This requirement was not explicitly specified in the C90 Standard.

C++

> *For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types.* 3.9.1p1

The C++ Standard does not include unsigned bit-fields in the above requirement, as C does. However, it is likely that implementations will follow the C requirement.

572 Values stored in non-bit-field objects of any other object type consist of $n \times$ `CHAR_BIT` bits, where n is the size of an object of that type, in bytes.

C90

This level of detail was not specified in the C90 Standard (and neither were any of the other details in this paragraph).

C++

> *The object representation of an object of type `T` is the sequence of N* **`unsigned char`** *objects taken up by the object of type `T`, where N equals `sizeof(T)`.* 3.9p4

That describes the object representation. But what about the value representation?

> *The value representation of an object is the set of bits that hold the value of type `T`. For POD types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.[37]* 3.9p4

This does not tie things down as tightly as the C wording. In fact later on we have:

3.9.1p1

For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types.

QED.

---

value copied using unsigned char

The value may be copied into an object of type **unsigned char [*n*]** (e.g., by **memcpy**); 573

C90

This observation was first made by the response to DR #069.

---

bit-field value is m bits

Values stored in bit-fields consist of *m* bits, where *m* is the size specified for the bit-field. 575

C++

9.6p1 *The `constant-expression` may be larger than the number of bits in the object representation (3.9) of the bit-field's type; in such cases the extra bits are used as padding bits and do not participate in the value representation (3.9) of the bit-field.*

This specifies the object representation of bit-fields. The C++ Standard does not say anything about the representation of values stored in bit-fields.

C++ allows bit-fields to contain padding bits. When porting software to a C++ translator, where the type **int** has a smaller width (e.g., 16 bits), there is the possibility that some of the bits will be treated as padding bits on the new host. In:

```
struct T {
          unsigned int m1:18;
         };
```

the member `m1` will have 18 value bits when the type **unsigned int** has a precision of 32, but only 16 value bits when **unsigned int** has a precision of 16.

Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations. 577

C++

3.9p3 *For any POD type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, if the value of `obj1` is copied into `obj2`, using the `memcpy` library function, `obj2` shall subsequently hold the same value as `obj1`.*

This handles the first case above. The C++ Standard says nothing about the second value compare case.

Certain object representations need not represent a value of the object type. 578

C90

This observation was not explicitly made in the C90 Standard.

C++

3.9p4

The value representation of an object is the set of bits that hold the value of type T*. For POD types, the value representation is a set of bits in the object representation that determines a value, which is one discrete element of an implementation-defined set of values.*[37)]

37) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Languages C. Footnote 37

By implication this is saying what C says. It also explicitly specifies that these representation issues are implementation-defined.

---

579 If the stored value of an object has such a representation and is read by an lvalue expression that does not have character type, the behavior is undefined. trap representation reading is undefined behavior

C90

The C90 Standard specified that reading an uninitialized object was undefined behavior. But, it did not specify undefined behaviors for any other representations.

C++

The C++ Standard does not explicitly specify any such behavior.

---

580 If such a representation is produced by a side effect that modifies all or any part of the object by an lvalue expression that does not have character type, the behavior is undefined.[41)]

C90

The C90 Standard did not explicitly specify this behavior.

C++

The C++ Standard does not explicitly discuss this issue.

---

581 Such a representation is called a *trap representation*. trap representation

C90

This term was not defined in the C90 Standard.

C++

Trap representations in C++ only apply to floating-point types.

---

582 40) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral powers of 2, except perhaps the bit with the highest position. footnote 40

C90

Integer type representation issues were discussed in DR #069.

---

584 A byte contains **CHAR_BIT** bits, and the values of type **unsigned char** range from 0 to $2^{\texttt{CHAR\_BIT}} - 1$. unsigned char value range

C++

The C++ Standard includes the C library by reference, so a definition of CHAR_BIT will be available in C++.

Unsigned integers, declared **unsigned***, shall obey the laws of arithmetic modulo* 2^n *where* n *is the number of bits in the value representation of that particular size of integer.*[41)] 3.9.1p4

From which it can be deduced that the C requirement holds true in C++.

footnote 41

41) Thus, an automatic variable can be initialized to a trap representation without causing undefined behavior, but the value of the variable cannot be used until a proper value is stored in it. 585

C90

The C90 Standard did not discuss trap representation and this possibility was not discussed.

C++

The C++ Standard does not make this observation about possible implementation behavior.

value stored in structure
value stored in union

When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.[42] 586

C90

The sentences:

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

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

appeared in C90, but does not appear in C99.

If a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is implementation-defined in C90 and unspecified in C99.

C++

This specification was added in C99 and is not explicitly specified in the C++ Standard.

~~The values of padding bytes shall not affect whether the value of such an object is a trap representation.~~ The value of a structure or union object is never a trap representation, even though the value of a member of a structure or union object may be a trap representation. 587

C90

This requirement is new in C99.

C++

This wording was added in C99 and is not explicitly specified in the C++ Standard.

Where an operator is applied to a value that has more than one object representation, which object representation is used shall not affect the value of the result.[43] 590

C90

This requirement was not explicitly specified in the C90 Standard.

C++

This requirement was added in C99 and is not explicitly specified in the C++ Standard (although the last sentence of 3.9p4 might be interpreted to imply this behavior).

object representation more than one

Where a value is stored in an object using a type that has more than one object representation for that value, it is unspecified which representation is used, but a trap representation shall not be generated. 591

C90

The C90 Standard was silent on the topic of multiple representations of object types.

C++

This requirement is not explicitly specified in the C++ Standard.

6.2.6.2 Integer types

593 For unsigned integer types other than **unsigned char**, the bits of the object representation shall be divided into two groups: value bits and padding bits (there need not be any of the latter). unsigned integer types object representation

C90

Explicit calling out of a division of the bits in the representation of an unsigned integer representation is new in C99.

C++

Like C90 the grouping of bits into value and padding bits is not explicitly specified in the C++ Standard (3.9p2, also requires that the type **unsigned char** not have any padding bits).

594 If there are N value bits, each bit shall represent a different power of 2 between 1 and 2^{N-1}, so that objects of that type shall be capable of representing values from 0 to 2^N-1 using a pure binary representation;

C90

These properties of unsigned integer types were not explicitly specified in the C90 Standard.

596 The values of any padding bits are unspecified.[44)] unsigned integer padding bit values

C90

Padding bits were not discussed in the C90 Standard, although they existed in some C90 implementations.

C++

This specification of behavior was added in C99 and is not explicitly specified in the C++ Standard.

599 there shall be exactly one sign bit. sign one bit

C90

This requirement was not explicitly specified in the C90 Standard.

C++

[Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types.] 3.9.1p7

These three representations all have exactly one sign bit.

600 Each bit that is a value bit shall have the same value as the same bit in the object representation of the corresponding unsigned type (if there are M value bits in the signed type and N in the unsigned type, then $M \le N$). value bits signed/unsigned

C90

This requirement is new in C99.

C++

3.9.1p3

The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.

If the value representation is the same, the value bits will match up. What about the requirement on the number of bits?

3.9.1p3 *. . . each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type[40]; that is, each signed integer type has the same object representation as its corresponding unsigned integer type.*

Combining these requirements with the representations listed in 3.9.1p7, we can deduce that C++ has the same restrictions on the relative number of value bits in signed and unsigned types.

---

footnote 42

42)Thus, for example, structure assignment ~~may be implemented element-at-a-time or via `memcpy`.~~ need not copy any padding bits. 601

C90

The C90 Standard did not explicitly specify that padding bits need not be copied.

C++

Footnote 36 *36) By using, for example, the library functions (17.4.1.2) `memcpy` or `memmove`.*

The C++ Standard does not discuss details of structure object assignment for those constructs that are supported by C. However, it does discuss this issue (12.8p8) for copy constructors, a C++ construct.

---

footnote 43

43) It is possible for objects `x` and `y` with the same effective type `T` to have the same value when they are accessed as objects of type `T`, but to have different values in other contexts. 602

C90

This observation was not pointed out in the C90 Standard.

C++

The C++ Standard does not make a general observation on this issue. However, it does suggest that such behavior might occur as a result of the **`reinterpret_cast`** operator (5.2.10p3).

---

footnote 44

44) Some combinations of padding bits might generate trap representations, for example, if one padding bit is a parity bit. 606

C90

This footnote is new in C99.

C++

This wording was added in C99 and is not explicitly specified in the C++ Standard. Trap representations in C++ only apply to floating-point types.

---

arithmetic operation exceptional condition

Regardless, no arithmetic operation on valid values can generate a trap representation other than as part of an exceptional condition such as an overflow, and this cannot occur with unsigned types. 607

C++

This discussion on trap representations was added in C99 and is not explicitly specified in the C++ Standard.

608 All other combinations of padding bits are alternative object representations of the value specified by the value bits.

C++

This discussion of padding bits was added in C99 and is not explicitly specified in the C++ Standard.

609 If the sign bit is zero, it shall not affect the resulting value.

C90

This requirement was not explicitly specified in the C90 Standard.

C++

> *[Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types.]* 3.9.1p7

In these three representations a sign bit of zero does not affect the resulting value.

610 If the sign bit is one, the value shall be modified in one of the following ways: sign bit representation

C90

This requirement was not explicitly specified in the C90 Standard.

611 — the corresponding value with sign bit 0 is negated (*sign and magnitude*); sign and magnitude

C++

Support for sign and magnitude is called out in 3.9.1p7, but the representational issues are not discussed.

612 — the sign bit has the value -(2^N) (*two's complement*); two's complement

C++

Support for two's complement is called out in 3.9.1p7, but the representational issues are not discussed.

613 — the sign bit has the value $-(2^N - 1)$ (*one's complement*). one's complement

C++

Support for one's complement is called out in 3.9.1p7, but the representational issues are not discussed.

614 Which of these applies is implementation-defined, as is whether the value with sign bit 1 and all value bits zero (for the first two), or with sign bit and all value bits 1 (for one's complement), is a trap representation or a normal value.

C90

The choice of representation for signed integer types was not specified as implementation-defined in C90 (although annex G.3.5 claims otherwise). The C90 Standard said nothing about possible trap representations.

C++

The following suggests that the behavior is unspecified.

> *The representations of integral types shall define values by use of a pure binary numeration system[44]. [Example: this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types.]* 3.9.1p7

Trap representations in C++ only apply to floating-point types.

negative zero

In the case of sign and magnitude and one's complement, if this representation is a normal value it is called a *negative zero*. 615

C90

The C90 Standard supported hosts that included a representation for negative zero, however, the term *negative zero* was not explicitly defined.

C++

The term *negative zero* does not appear in the C++ Standard.

negative zero only generated by

If the implementation supports negative zeros, they shall be generated only by: 616

C90

The properties of negative zeros were not explicitly discussed in the C90 Standard.

C++

This requirement was added in C99; negative zeros are not explicitly discussed in the C++ Standard.

negative zero storing

It is unspecified whether these cases actually generate a negative zero or a normal zero, and whether a negative zero becomes a normal zero when stored in an object. 620

C90

This unspecified behavior was not called out in the C90 Standard, which did not discuss negative integer zeros.

C++

Negative zero is not discussed in the C++ Standard.

If the implementation does not support negative zeros, the behavior of the `&`, `|`, `^`, `~`, `<<`, and `>>` operators with arguments that would produce such a value is undefined. 621

C90

This undefined behavior was not explicitly specified in the C90 Standard.

C++

This specification was added in C99 and is not explicitly specified in the C++ Standard.

object representation same padding signed/unsigned

A valid (non-trap) object representation of a signed integer type where the sign bit is zero is a valid object representation of the corresponding unsigned type, and shall represent the same value. 623

positive signed integer type subrange of equivalent unsigned type

C90

There was no such requirement on the object representation in C90, although this did contain the C99 requirement on the value representation.

C++

3.9.1p3

> *. . . ; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.*

For any integer type, the object representation where all the bits are zero shall be a representation of the value zero in that type. 624

C90

The C90 Standard did not specify this requirement.

---

625 The *precision* of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits. precision integer type

C90

The definition of this term is new in C99.

C++

The term *precision* was added in C99 and is only defined in C++ for floating-point types.

---

626 The *width* of an integer type is the same but including any sign bit; width integer type

C90

The definition of this term is new in C99.

C++

The term *width* was added in C99 and is not defined in the C++ Standard.

6.2.7 Compatible type and composite type

---

631 Two types have *compatible type* if their types are the same. compatible type if same type

C++

The C++ Standard does not define the term *compatible type*. It either uses the term *same type* or *different type*. The terms *layout-compatible* and *reference-compatible* are defined by the C++ Standard. The specification of layout-compatible structure (9.2p14) and layout compatible union (9.2p15) is based on the same set of rules as the C cross translation unit compatibility rules. The purpose of layout compatibility deals with linking objects written in other languages; C is explicitly called out as one such language.

---

633 Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: compatible separate translation units

C90

> *Moreover, two structure, union, or enumerated types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types;*

There were no requirements specified for tag names in C90. Since virtually no implementation performs this check, it is unlikely that any programs will fail to link when using a C99 implementation.

C++

The following paragraph applies when both translation units are written in C++.

> *There can be more than one definition of a class type (clause 9), enumeration type (7.2), inline function with external linkage (7.1.2), class template (clause 14), non-static function template (14.5.5), static data member of a class template (14.5.1.3), member function template (14.5.1.1), or template specialization for which some template parameters are not specified (14.7, 14.5.4) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements.* 3.2p5

There is a specific set of rules for dealing with the case of one or more translation units being written in another language and others being written in C++:

7.5p2

> *Linkage (3.5) between C++ and non-C++ code fragments can be achieved using a `linkage-specification`:*
>
> . . .
>
> *[Note: . . . The semantics of a language linkage other than C++ or C are implementation-defined.]*

which appears to suggest that it is possible to make use of defined behavior when building a program image from components translated using both C++ and C translators.

The C++ Standard also requires that:

7.1.5.3p3
> *The `class-key` or **`enum`** keyword present in the `elaborated-type-specifier` shall agree in kind with the declaration to which the name in the `elaborated-type-specifier` refers.*

---

tag declared with same

If one is declared with a tag, the other shall be declared with the same tag. 634

C90

This requirement is not specified in the C90 Standard.

Structures declared using different tags are now considered to be different types.

xfile.c

```
#include <stdio.h>

extern int WG14_N685(struct tag1 *, struct tag1 *);

struct tag1 {
            int m1,
                m2;
            } st1;

void f(void)
{
if (WG14_N685(&st1, &st1))
   {
   printf("optimized\n");
   }
else
   {
   printf("unoptimized\n");
   }
}
```

yfile.c

```
struct tag2 {
            int m1,
                m2;
            };
struct tag3 {
            int m1,
                m2;
            };

int WG14_N685(struct tag2 *pst1,
              struct tag3 *pst2)
{
pst1->m1 = 2;
pst2->m1 = 0;   /* alias? */

return pst1->m1;
}
```

An optimizing translator might produce **optimized** as the output of the program, while the same translator with optimization turned off might produce **unoptimized** as the output. This is because translation unit `y.c` defines `func` with two parameters each as pointers to different structures, and translation unit `x.c` calls `WG14_N685func` but passes the address of the same structure for each argument.

---

636 there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types, and such that if one member of a corresponding pair is declared with a name, the other member is declared with the same name.

C90

> *. . . if they have the same number of members, the same member names, and compatible member types;*

The C90 Standard is lax in that it does not specify any correspondence for members defined in different structure types, their names and associated types.

C++

> *— each definition of `D` shall consist of the same sequence of tokens; and* 3.2p5
>
> *— in each definition of `D`, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of `D`, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a **`const`** object with internal or no linkage if the object has the same integral or enumeration type in all definitions of `D`, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of `D`; and*

The C Standard specifies an effect, compatible types. The C++ Standard specifies an algorithm, the same sequence of tokens (not preprocessing tokens), which has several effects. The following source files are strictly conforming C, but undefined behavior in C++.

file_1.c

```
extern struct {
              short s_mem1;
              } glob;
```

file_2.c

```
extern struct {
              short int s_mem1;
              } glob;
```

> *Two POD-struct (clause 9) types are layout-compatible if they have the same number of members, and corresponding members (in order) have layout-compatible types (3.9).* 9.2p14

> *Two POD-union (clause 9) types are layout-compatible if they have the same number of members, and corresponding members (in any order) have layout-compatible types (3.9).* 9.2p15

Layout compatibility plays a role in interfacing C++ programs to other languages and involves types only. The names of members plays no part.

members corresponding

For two structures, corresponding members shall be declared in the same order. 637

C90

. . . for two structures, the members shall be in the same order;

The C90 Standard is lax in that it does not specify how a correspondence is formed between members defined in different structure definitions. The following two source files could have been part of a strictly conforming program in C90. In C99 the behavior is undefined and, if the output depends on `glob`, the program will not be strictly conforming.

file_1.c

```
extern struct {
              short s_mem1;
              int i_mem2;
              } glob;
```

file_2.c

```
extern struct {
              int i_mem2;
              short s_mem1;
              } glob;
```

While the C90 Standard did not require an ordering of corresponding member names, developer expectations do. A diagnostic, issued by a C99 translator, for a declaration of the same object as a structure type with differing member orders, is likely to be welcomed by developers.

For two enumerations, corresponding members shall have the same values. 639

C90

. . . for two enumerations, the members shall have the same values.

The C90 Standard is lax in not explicitly specifying that the members with the same names have the same values.

C++

3.2p5 *— each definition of `D` shall consist of the same sequence of tokens; and*

The C++ requirement is stricter than C. In the following two translation units, the object `e_glob` are not considered compatible in C++:

file_1.c

```
extern enum {A = 1, B = 2} e_glob;
```

file_2.c

```
extern enum {B= 2, A = 1} e_glob;
```

640 All declarations that refer to the same object or function shall have compatible type; same object have compatible types same function have compatible types

C++

> — *each definition of* `D` *shall consist of the same sequence of tokens; and* 3.2p5
>
> — *in each definition of* `D`*, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of* `D`*, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a* **`const`** *object with internal or no linkage if the object has the same integral or enumeration type in all definitions of* `D`*, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of* `D`*; and*

> *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4).* 3.5p10

The C++ Standard is much stricter in requiring that the types be identical. The **`int/enum`** example given above would not be considered compatible in C++. If translated and linked with each other the following source files are strictly conforming C, but undefined behavior in C++.

file_1.c

```
extern short es;
```

file_2.c

```
extern short int es = 2;
```

641 otherwise, the behavior is undefined.

C++

> *A violation of this rule on type identity does not require a diagnostic.* 3.5p10

The C++ Standard bows to the practical difficulties associated with requiring implementations to issue a diagnostic for this violation.

642 A *composite type* can be constructed from two types that are compatible; composite type

C++

One of the two types involved in creating composite types in C is not supported in C++ (function types that don't include prototypes) and the C++ specification for the other type (arrays) is completely different from C. 644 array composite type

Because C++ supports operator overloading type qualification of pointed-to types is a more pervasive issue than in C (where it only has to be handled for the conditional operator). The C++ Standard defines the concept of a *composite pointer type* (5.9p2). This specifies how a result type is constructed from pointers to qualified types, and the null pointer constant and other pointer types. conditional operator pointer to qualified types

644 — If one type is an array of known constant size, the composite type is an array of that size; array composite type

C90

> *If one type is an array of known size, the composite type is an array of that size;*

Support for arrays declared using a nonconstant size is new in C99.

C++

An incomplete array type can be completed. But the completed type is not called the composite type, and is regarded as a different type:

3.9p7
> *. . . ; the array types at those two points ("array of unknown bound of T" and "array of N T") are different types.*

The C++ Standard recognizes the practice of an object being declared with both complete and incomplete array types with the following exception:

3.5p10
> *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4).*

---

otherwise, if one type is a variable length array, the composite type is that type. 645

C90

Support for VLA types is new in C99.

C++

Variable length array types are new in C99. The C++ library defined container classes (23), but this is a very different implementation concept.

---

function composite type

— If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list. 646

C++

All C++ functions must be declared using prototypes. A program that contains a function declaration that does not include parameter information is assumed to take no parameters.

```
extern void f();

void g(void)
{
f(); // Refers to a function returning int and having no parameters
     /* Non-prototype function referenced */
}

void f(int p) /* Composite type formed, call in g linked to here */
              // A different function from int f()
              // Call in g does not refer to this function
{ /* ... */ }
```

— If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters. 647

C++

C++ allows functions to be overloaded based on their parameter types. An implementation must not form a composite type, even when the types might be viewed by a C programmer as having the same effect:

```
/*
 * A common, sloppy, coding practice.  Don't declare
 * the prototype to take enums, just use int.
 */
extern void f(int);

enum ET {E1, E2, E3};

void f(enum ET p) /* composite type formed, call in g linked to here */
                  // A different function from void f(int)
                  // Call in g does not refer here
{ /* ... */ }

void g(void)
{
f(E1); // Refers to a function void (int)
       /* Refers to definition of f above */
}
```

---

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

C++

The C++ Standard has no such rules to apply recursively.

---

649 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible,[47] if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type. prior declaration visible

C90

The wording in the C90 Standard:

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

was changed to its current form by the response to DR #011, question 1.

C++

> *Two names that are the same (clause 3) and that are declared in different scopes shall denote the same object, reference, function, type, enumerator, template or namespace if* 3.5p9
>
> *— both names have external linkage or else both names have internal linkage and are declared in the same translation unit; and*
>
> *— both names refer to members of the same namespace or to members, not by inheritance, of the same class; and*
>
> *— when both names denote functions, the function types are identical for purposes of overloading; and*

This paragraph applies to names declared in different scopes; for instance, file scope and block scope externals.

13p1

When two or more different declarations are specified for a single name in the same scope, that name is said to be overloaded. By extension, two declarations in the same scope that declare the same name but with different types are called overloaded declarations. Only function declarations can be overloaded; object and type declarations cannot be overloaded.

The following C++ requirement is much stricter than C. The types must be the same, which removes the need to create a composite type.

3.5p10 *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4). A violation of this rule on type identity does not require a diagnostic.*

The only composite type in C++ are composite pointer types (5.9p2). These are only used in relational operators (5.9p2), equality operators (5.10p2, where the term common type is used), and the conditional operator (5.16p6). C++ composite pointer types apply to the null pointer and possibly qualified pointers to **void**.

If declarations of the same function do not have the same type, the C++ link-time behavior will be undefined. Each function declaration involving different adjusted types will be regarded as referring to a different function.

```
extern void f(const int);
extern void f(int);        /* Conforming C, composite type formed */
                           // A second (and different) overloaded declaration
```

---

footnote 46

46) Two types need not be identical to be compatible. 650

C++

The term *compatible* is used in the C++ Standard for *layout-compatible* and *reference-compatible*. Layout compatibility is aimed at cross-language sharing of data structures and involves types only. The names of structure and union members, or tags, need not be identical. C++ reference types are not available in C.

EXAMPLE Given the following two file scope declarations: 652

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

The resulting composite type for the function is:

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

C++

The C++ language supports the overloading of functions. They are overloaded by having more than one declaration of a function with the same name and return type, but different parameter types. In C++ the two declarations in the example refer to different versions of the function `f`.

6.3 Conversions

---

implicit conversion
explicit conversion

This subclause specifies the result required from such an *implicit conversion*, as well as those that result from a cast operation (an *explicit conversion*). 654

C++

The C++ Standard defines a set of implicit and explicit conversions. Declarations contained in library headers also contain constructs that can cause implicit conversions (through the declaration of constructors) and support additional explicit conversions— for instance, the complex class.

The C++ language differs from C in that the set of implicit conversions is not fixed. It is also possible for user-defined declarations to create additional implicit and explicit conversions.

---

655 The list in 6.3.1.8 summarizes the conversions performed by most ordinary operators;

C++

Clause 4 'Standard conversions' and 5p9 define the conversions in the C++ Standard.

---

656 it is supplemented as required by the discussion of each operator in 6.5.

C++

There are fewer such supplements in the C++ Standard, partly due to the fact that C++ requires types to be the same and does not use the concept of compatible type.

C++ supports user-defined overloading of operators. Such overloading could change the behavior defined in the C++ Standard, however these definitions cannot appear in purely C source code.

---

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

C++

No such wording applied to the same types appears in the C++ Standard. Neither of the two uses of the C++ term *compatible* (layout-compatible, reference-compatible) discuss conversions.

6.3.1 Arithmetic operands

6.3.1.1 Boolean, characters, and integers

---

659 Every integer type has an *integer conversion rank* defined as follows: conversion rank

C90

The concept of integer conversion rank is new in C99.

C++

The C++ Standard follows the style of documenting the requirements used in the C90 Standard. The conversions are called out explicitly rather than by rank (which was introduced in C99). C++ supports operator overloading, where the conversion rules are those of a function call. However, this functionality is not available in C.

---

661 — The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision. rank signed integer vs less precision

C++

The relative, promotion, ordering of signed integer types defined by the language is called out explicitly in clause 5p9.

---

662 — The rank of **`long long int`** shall be greater than the rank of **`long int`**, which shall be greater than the rank of **`int`**, which shall be greater than the rank of **`short int`**, which shall be greater than the rank of **`signed char`**. rank standard integer types

C++

Clause 5p9 lists the pattern of the usual arithmetic conversions. This follows the relative orderings of rank given here (except that the types **`short int`** and **`signed char`** are not mentioned; nor would they be since the integral promotions would already have been applied to operands having these types).

---

rank standard integer relative to extended

— The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width. 664

C++

The C++ Standard specifies no requirements on how an implementation might extend the available integer types.

_Bool rank

— The rank of **`_Bool`** shall be less than the rank of all other standard integer types. 666

C++

3.9.1p6 *As described below, **`bool`** values behave as integral types.*

4.5p4 *An rvalue of type **`bool`** can be converted to an rvalue of type **`int`**, with **`false`** becoming zero and **`true`** becoming one.*

The C++ Standard places no requirement on the relative size of the type **`bool`** with respect to the other integer types. An implementation may choose to hold the two possible values in a single byte, or it may hold those values in an object that has the same width as type **`long`**.

---

rank extended integer relative to extended

— The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation-defined, but still subject to the other rules for determining the integer conversion rank. 668

C++

The C++ Standard does not specify any properties that must be given to user-defined classes that provide some form of extended integer type.

---

expression wherever an int may be used

The following may be used in an expression wherever an **`int`** or **`unsigned int`** may be used: 670

C90

The C90 Standard listed the types, while the C99 Standard bases the specification on the concept of rank.

*A **`char`**, a **`short int`**, or an **`int`** bit-field, or their signed or unsigned varieties, or an enumeration type, may be used in an expression wherever an **`int`** or **`unsigned int`** may be used.*

C++

C++ supports the overloading of operators; for instance, a developer-defined definition can be given to the binary **`+`** operator, when applied to operands having type **`short`**. Given this functionality, this C sentence cannot be said to universally apply to programs written in C++. It is not listed as a difference because it requires use of C++ functionality for it to be applicable. The implicit conversion sequences are specified in clause 13.3.3.1. When there are no overloaded operators visible (or to be exact no overloaded operators taking arithmetic operands, and no user-defined conversion involving arithmetic types), the behavior is the same as C.

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

C++

*An rvalue of type **char**, **signed char**, **unsigned char**, **short int**, or **unsigned short int** can be converted to an rvalue of type **int** if **int** can represent all the values of the source type; otherwise, the source rvalue can be converted to an rvalue of type **unsigned int**.* 4.5p1

*An rvalue of type **wchar_t** (3.9.1) or an enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: **int**, **unsigned int**, **long**, or **unsigned long**.* 4.5p2

*An rvalue of type **bool** can be converted to an rvalue of type **int**, with **false** becoming zero and **true** becoming one.* 4.5p4

The key phrase here is *can be*, which does not imply that they *shall be*. However, the situations where these conversions might not apply (e.g., operator overloading) do not involve constructs that are available in C. For binary operators the *can be* conversions quoted above become *shall be* requirements on the implementation (thus operands with rank less than the rank of **int** are supported in this context):

Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the usual arithmetic conversions, which are defined as follows: 5p9

— Otherwise, the integral promotions (4.5) shall be performed on both operands.[54)]

*54) As a consequence, operands of type **bool**, **wchar_t**, or an enumerated type are converted to some integral type.* Footnote 54

The C++ Standard does not appear to contain explicit wording giving this permission for other occurrences of operands (e.g., to unary operators). However, it does not contain wording prohibiting the usage (the wording for the unary operators invariably requires the operand to have an arithmetic or scalar type).

672 — A bit-field of type **_Bool**, **int**, **signed int**, or **unsigned int**. bit-field in expression

C90

Support for bit-fields of type **_Bool** is new in C99.

C++

*An rvalue for an integral bit-field (9.6) can be converted to an rvalue of type **int** if **int** can represent all the values of the bit-field; otherwise, it can be converted to **unsigned int** if **unsigned int** can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.* 4.5p3

C does not support the definition of bit-fields that are larger than type **int**, or bit-fields having an enumerated type.

integer promotions

These are called the *integer promotions*.[48] 675

C90

These are called the integral promotions.[27]

C++

The C++ Standard uses the C90 Standard terminology (and also points out, 3.9.1p7, "A synonym for integral type is *integer type*.").

All other types are unchanged by the integer promotions. 676

C++

This is not explicitly specified in the C++ Standard. However, clause 4.5, Integral promotions, discusses no other types, so the statement is also true in C++

6.3.1.2 Boolean type

_Bool converted to

When any scalar value is converted to **`_Bool`**, the result is 0 if the value compares equal to 0; 680

C90

Support for the type **`_Bool`** is new in C99.

C++

4.12p1 *An rvalue of arithmetic, enumeration, pointer, or pointer to member type can be converted to an rvalue of type* ***`bool`****. A zero value, null pointer value, or null member pointer value is converted to* ***`false`****;*

The value of **`false`** is not defined by the C++ Standard (unlike **`true`**, it is unlikely to be represented using any value other than zero). But in contexts where the integer conversions are applied:

4.7p4 *. . . the value* ***`false`*** *is converted to zero . . .*

otherwise, the result is 1. 681

C++

4.12p1 *. . . ; any other value is converted to* ***`true`****.*

The value of **`true`** is not defined by the C++ Standard (implementations may choose to represent it internally using any nonzero value). But in contexts where the integer conversions are applied:

4.7p4 *. . . the value* ***`true`*** *is converted to one.*

6.3.1.3 Signed and unsigned integers

682 When a value with integer type is converted to another integer type other than **_Bool**, if the value can be represented by the new type, it is unchanged.

C90

Support for the type **_Bool** is new in C99, and the C90 Standard did not need to include it as an exception.

683 Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.[49] unsigned integer conversion to

C90

Otherwise: if the unsigned integer has greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned integer; the value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.[28]

When a value with integral type is demoted to an unsigned integer with smaller size, the result is the nonnegative remainder on division by the number one greater than the largest unsigned number that can be represented in the type with smaller size.

The C99 wording is a simpler way of specifying the C90 behavior.

685 either the result is implementation-defined or an implementation-defined signal is raised. signed integer conversion implementation-defined

C90

The specification in the C90 Standard did not explicitly specify that a signal might be raised. This is because the C90 definition of implementation-defined behavior did not rule out the possibility of an implementation raising a signal. The C99 wording does not permit this possibility, hence the additional permission given here.

C++

. . . ; otherwise, the value is implementation-defined. 4.7p3

The C++ Standard follows the wording in C90 and does not explicitly permit a signal from being raised in this context because this behavior is considered to be within the permissible range of implementation-defined behaviors.

6.3.1.4 Real floating and integer

686 When a finite value of real floating type is converted to an integer type other than **_Bool**, the fractional part is discarded (i.e., the value is truncated toward zero). floating-point converted to integer

C90

Support for the type **_Bool** is new in C99.

688 When a value of integer type is converted to a real floating type, if the value being converted can be represented exactly in the new type, it is unchanged. integer conversion to floating

C++

4.9p2

An rvalue of an integer type or of an enumeration type can be converted to an rvalue of a floating point type. The result is exact if possible.

Who decides what is possible or if it can be represented exactly? A friendly reading suggests that the meaning is the same as C99.

---

footnote 48

48) The integer promotions are applied only: as part of the usual arithmetic conversions, to certain argument expressions, to the operands of the unary +, -, and ~ operators, and to both operands of the shift operators, as specified by their respective subclauses. 690

C++

In C++, integral promotions are applied also as part of the usual arithmetic conversions, the operands of the unary +, -, and ~ operators, and to both operands of the shift operators. C++ also performs integer promotions in contexts not mentioned here, as does C.

---

footnote 49

49) The rules describe arithmetic on the mathematical value, not the value of a given type of expression. 691

C90

This observation was not made in the C90 Standard (but was deemed to be implicitly true).

C++

The C++ Standard does not make this observation.

---

footnote 50

50) The remaindering operation performed when a value of integer type is converted to unsigned type need not be performed when a value of real floating type is converted to unsigned type. 692

C++

4.9p1 *An rvalue of a floating point type can be converted to an rvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type.*

The conversion behavior, when the result cannot be represented in the destination type is undefined in C++ and unspecified in C.

Thus, the range of portable real floating values is $(-1, \texttt{U}type\texttt{\_MAX} + 1)$. 693

C++

The C++ Standard does not make this observation.

If the value being converted is outside the range of values that can be represented, the behavior is undefined. 694

C++

4.9p2 *Otherwise, it is an implementation-defined choice of either the next lower or higher representable value.*

The conversion behavior, when the result is outside the range of values that can be represented in the destination type, is implementation-defined in C++ and undefined in C.

6.3.1.5 Real floating types

695 When a **float** is promoted to **double** or **long double**, or a **double** is promoted to **long double**, its value is unchanged (if the source value is represented in the precision and range of its type). float promoted to double or long double

C++

*The type **double** provides at least as much precision as **float**, and the type **long double** provides at least as much precision as **double**. The set of values of the type **float** is a subset of the set of values of the type **double**; the set of values of the type **double** is a subset of the set of values of the type **long double**.* 3.9.1p8

This only gives a relative ordering on the available precision. It does not say anything about promotion leaving a value unchanged.

*An rvalue of type **float** can be converted to an rvalue of type **double**. The value is unchanged.* 4.6p1

There is no equivalent statement for type **double** to **long double** promotions. But there is a general statement about conversion of floating-point values:

An rvalue of floating point type can be converted to an rvalue of another floating point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. 4.8p1

Given that (1) the set of values representable in a floating-point type is a subset of those supported by a wider floating-point type (3.9.1p8); and (2) when a value is converted to the wider type, the exact representation is required to be used (by 4.8p1)— the value must be unchanged.

696 When a **double** is demoted to **float**, a **long double** is demoted to **double** or **float**, or a value being represented in greater precision and range than required by its semantic type (see 6.3.1.8) is explicitly converted <iso_delete>to its semantic type</iso_delete> (including to its own type), if the value being converted can be represented exactly in the new type, it is unchanged. double demoted to another floating type

C90

This case is not specified in the C90 Standard.

6.3.1.6 Complex types

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

C90

Support for complex types is new in C99.

C++

The C++ Standard does not provide a specification for how the conversions are to be implemented.

6.3.1.7 Real and complex

700 When a value of real type is converted to a complex type, the real part of the complex result value is determined by the rules of conversion to the corresponding real type and the imaginary part of the complex result value is a positive zero or an unsigned zero. real type converted to complex

C90

Support for complex types is new in C99.

C++

The constructors for the complex specializations, 26.2.3, take two parameters, corresponding to the real and imaginary part, of the matching floating-point type. The default value for the imaginary part is specified as 0.0.

When a value of complex type is converted to a real type, the imaginary part of the complex value is discarded 701
and the value of the real part is converted according to the conversion rules for the corresponding real type.

C++

In C++ the conversion has to be explicit. The member functions of the complex specializations (26.2.3) return a value that has the matching floating-point type.

6.3.1.8 Usual arithmetic conversions

---

common real type

The purpose is to determine a *common real type* for the operands and result. 703

C90

The term *common real type* is new in C99; the equivalent C90 term was *common type*.

---

arithmetic conversions type domain unchanged

For the specified operands, each operand is converted, without change of type domain, to a type whose 704
corresponding real type is the common real type.

C90

Support for type domains is new in C99.

C++

The term *type domain* is new in C99 and is not defined in the C++ Standard.

The template class `complex` contain constructors that can be used to implicitly convert to the matching complex type. The operators defined in these templates all return the appropriate complex type.

C++ converts all operands to a complex type before performing the operation. In the above example the C result is $6.0 + \infty i$, while the C++ result is $NaN + \infty i$.

---

arithmetic conversions result type

Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, 705
whose type domain is the type domain of the operands if they are the same, and complex otherwise.

C++

The `complex` specializations (26.2.3) define conversions for **`float`**, **`double`** and **`long double`** to complex classes. A number of the constructors are defined as explicit, which means they do not happen implicitly, they can only be used explicitly. The effect is to create a different result type in some cases.

footnote 51 719

In C++, if the one operand does not have a complex type, it is converted to the corresponding complex type, and the result type is the same as the other operand having complex type. See footnote 51.

---

arithmetic conversions integer types

Then the following rules are applied to the promoted operands: 711

C++

The rules in the C++ Standard appear in a bulleted list of types with an implied sequential application order.

---

operand same type no further conversion

If both operands have the same type, then no further conversion is needed. 712

C90

For language lawyers only: A subtle difference in requirements exists between the C90 and C99 Standard (which in practice would have been optimized away by implementations). The rules in the C90 wording were ordered such that when two operands had the same type, except when both were type **`int`**, a conversion was required. So the type **unsigned long** needed to be converted to an **unsigned long**, or a **long** to a **long**, or an **unsigned int** to an **unsigned int**.

---

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

C90

Support for complex types is new in C99

C++

The conversion sequence is different in C++. In C++ the operand having type **float** will be converted to `complexfloat` prior to the addition operation. arithmetic conversions float

```
#include <complex.h> // the equivalent C++ header

float complex fc; // std::complex<float> fc;  this is the equivalent C++ declaration
double d;

void f(void)
{
fc + d   /* Result has type double complex. */
         // Result has type complex<float>.
       ;
}
```

6.3.2 Other operands

6.3.2.1 Lvalues, arrays, and function designators

---

721 An *lvalue* is an expression with an object type or an incomplete type other than **void**;[53)] lvalue

C90

> *An lvalue is an expression (with an object type or an incomplete type other than **void**) that designates an object.31)*

The C90 Standard required that an lvalue designate an object. An implication of this requirement was that some constraint requirements could only be enforced during program execution (e.g., the left operand of an assignment operator must be an lvalue). The Committee intended that constraint requirements be enforceable during translation. 1289 assignment operator modifiable lvalue

Technically this is a change of behavior between C99 and C90. But since few implementations enforced this requirement during program execution, the difference is unlikely to be noticed.

C++

> *An lvalue refers to an object or function.* 3.10p2

Incomplete types, other than **void**, are object types in C++, so all C lvalues are also C++ lvalues. 475 object types

The C++ support for a function lvalue involves the use of some syntax that is not supported in C.

3.10p3

As another example, the function

```
int& f();
```

yields an lvalue, so the call `f()` *is an lvalue expression.*

---

if an lvalue does not designate an object when it is evaluated, the behavior is undefined. 722

C90

In the C90 Standard the definition of the term *lvalue* required that it designate an object. An expression could not be an lvalue unless it designated an object.

C++

In C++ the behavior is not always undefined:

3.8p6 *Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any lvalue which refers to the original object may be used but only in limited ways. Such an lvalue refers to allocated storage (3.7.3.2), and using the properties of the lvalue which do not depend on its value is well-defined.*

---

particular type

When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. 723

C++

The situation in C++ is rather more complex:

1.8p1 *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.] 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.*

---

modifiable lvalue

A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type. 724

C++

The term *modifiable lvalue* is used by the C++ Standard, but understanding what this term might mean requires joining together the definitions of the terms *lvalue* and *modifiable*:

3.10p10

An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances.

If an expression can be used to modify the object to which it refers, the expression is called modifiable. 3.10p14

There does not appear to be any mechanism for modifying objects having an incomplete type.

Objects of array types cannot be modified, see 3.10. 8.3.4p5

This is a case where an object of a given type cannot be modified and follows the C requirement.

. . . ; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path. 7.1.5.1p3

The C++ wording is based on access paths rather than the C method of enumerating the various cases. However, the final effect is the same.

---

725 Except when it is the operand of the **`sizeof`** operator, the unary **`&`** operator, the `++` operator, the `--` operator, or the left operand of the **`.`** operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue). lvalue converted to value

C++

Quite a long chain of deduction is needed to show that this requirement also applies in C++. The C++ Standard uses the term *rvalue* to refer to the particular value that an lvalue is converted into.

Whenever an lvalue appears in a context where an rvalue is expected, the lvalue is converted to an rvalue; 3.10p7

Whenever an lvalue expression appears as an operand of an operator that expects an rvalue for that operand, the lvalue-to-rvalue (4.1), . . . standard conversions are applied to convert the expression to an rvalue. 5p8

The C wording specifies that lvalues are converted unless they occur in specified contexts. The C++ wording specifies that lvalues are converted in a context where an rvalue is expected. Enumerating the cases where C++ expects an rvalue we find:

The lvalue-to-rvalue (4.1), . . . standard conversions are not applied to the operand of ***`sizeof`****.* 5.3.4p4

What is the behavior for the unary **`&`** operator?

There are some contexts where certain conversions are suppressed. For example, the lvalue-to-rvalue conversion is not done on the operand of the unary **`&`** *operator.* 4p5

However, this is a *Note:* and has no normative status. There is no mention of any conversions in 5.3.1p2-5, which deals with the unary **`&`** operator.

In the case of the postfix `++` and `--` operators we have:

5.2.6p1

The operand shall be a modifiable lvalue. . . . The result is an rvalue.

In the case of the prefix `++` and `--` operators we have:

5.3.2p1 *The operand shall be a modifiable lvalue. . . . The value is the new value of the operand; it is an lvalue.*

So for the postfix case, there is an lvalue-to-rvalue conversion, although this is never explicitly stated and in the prefix case there is no conversion.

The C case is more restrictive than C++, which requires a conforming implementation to successfully translate:

```
extern int i;

void f(void)
{
++i = 4; // Well-formed
         /* Constraint violation */
}
```

For the left operand of the `.` operator we have:

5.2.5p4 *If `E1` is an lvalue, then `E1.E2` is an lvalue.*

The left operand is not converted to an rvalue. For the left operand of an assignment operator we have:

5.17p1 *All require a modifiable lvalue as their left operand, . . . ; the result is an lvalue.*

The left operand is not converted to an rvalue. And finally for the array type:

4.1p1 *An lvalue (3.10) of a non-function, non-array type `T` can be converted to an rvalue.*

An lvalue having an array type cannot be converted to an rvalue (i.e., the C++ Standard contains no other wording specifying that an array can be converted to an rvalue).

In two cases the C++ Standard specifies that lvalue-to-rvalue conversions are not applied: Clause 5.18p1 left operand of the comma operator and Clause 6.2p1 the expression in an expression statement. In C the values would be discarded in both of these cases, so there is no change in behavior. In the following cases C++ performs a lvalue-to-rvalue conversion (however, the language construct is not relevant to C): Clause 8.2.8p3 Type identification; 5.2.9p4 static cast; 8.5.3p5 References.

---

lvalue value is unqualified

If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue; 726

C++

The value being referred to in C is what C++ calls an *rvalue*.

4.1p1 *If T is a non-class type, the type of the rvalue is the cv-unqualified version of T. Otherwise, the type of the rvalue is T.*[49)]

Footnote 49 *49) In C++ class rvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types.*

Class rvalues are objects having a structure or union type in C.

728 If the lvalue has an incomplete type and does not have array type, the behavior is undefined.

C++

An lvalue (3.10) of a non-function, non-array type `T` *can be converted to an rvalue. If* `T` *is an incomplete type, a program that necessitates this conversion is ill-formed.* 4.1p1

An lvalue or rvalue of type "array of N `T`*" or "array of unknown bound of* `T`*" can be converted to an rvalue of type "pointer to* `T`*."* 4.2p1

The C behavior in this case is undefined; in C++ the conversion is ill-formed and a diagnostic is required.

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

C90

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

The C90 Standard says " . . . , an lvalue that has type "array of type" is converted . . . ", while the C99 Standard says " . . . , an expression that has type . . . ". It is possible to create a non-lvalue array. In these cases the behavior has changed. In C99 the expression `(g?x:y).m1[1]` is no longer a constraint violation (C90 footnote 50, "A conditional expression does not yield an lvalue").

In the following, C90 requires that the size of the pointer type be output, while C99 requires that the size of the array be output.

```
#include <stdio.h>

struct {
       int m1@lsquare[]2@rsquare[];
       } x, y;
int g;

int main(void)
{
printf("size=%ld\n", sizeof((g?x:y).m1));
}
```

C++

The . . . , array-to-pointer (4.2), . . . standard conversions are not applied to the operand of **`sizeof`**. 5.3.3p4

4.2p1

An lvalue or rvalue of type "array of N T*" or "array of unknown bound of* T*" can be converted to an rvalue of type "pointer to* T*." The result is a pointer to the first element of the array.*

4.2p2 *A string literal . . . can be converted . . . This conversion is considered only when there is an explicit appropriate pointer target type, and not when there is a general need to convert from an lvalue to an rvalue.*

When is there an explicit appropriate pointer target type? Clause 5.2.1 Subscripting, requires that one of the operands have type *pointer to T*. A character string literal would thus be converted in this context.

5.17p3 *If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand.*

8.3.5p3 *After determining the type of each parameter, any parameter of type "array of* T*" or "function returning* T*" is adjusted to be "pointer to* T*" or "pointer to function returning* T*," respectively.*

Clause 5.3.1p2 does not say anything about the conversion of the operand of the unary **&** operator. Given that this operator requires its operand to be an lvalue not converting an lvalue array to an rvalue in this context would be the expected behavior.

There may be other conversions, or lack of, that are specific to C++ constructs that are not supported in C.

---

array object register storage class

If the array object has register storage class, the behavior is undefined. 730

C90

This behavior was not explicitly specified in the C90 Standard.

C++

7.1.1p3 *A* **`register`** *specifier has the same semantics as an* **`auto`** *specifier together with a hint to the implementation that the object so declared will be heavily used.*

Source developed using a C++ translator may contain declarations of array objects that include the **`register`** storage class. The behavior of programs containing such declarations will be undefined if processed by a C translator.

---

function designator

A *function designator* is an expression that has function type. 731

C++

This terminology is not used in the C++ Standard.

---

function designator converted to type

Except when it is the operand of the **`sizeof`** operator[54)] or the unary **&** operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*". 732

C++

5.3.3p4

The . . . , and function-to-pointer (4.3) standard conversions are not applied to the operand of **`sizeof`**.

The result of the unary **`&`** *operator is a pointer to its operand. The operand shall be an lvalue or a* `qualified-id`*. In the first case, if the type of the expression is "*`T`*," the type of the result is "pointer to* `T`*."* 5.3.1p2

While this clause does not say anything about the conversion of the operand of the unary **`&`** operator, given that this operator returns a result whose type is "pointer to T", not converting it prior to the operator being applied would be the expected behavior. What are the function-to-pointer standard conversions?

An lvalue of function type `T` *can be converted to an rvalue of type "pointer to* `T`*."* 4.3p1

Whenever an lvalue expression appears as an operand of an operator that expects an rvalue for that operand, . . . , or function-to-pointer (4.3) standard conversions are applied to convert the expression to an rvalue. 5p8

In what contexts does an operator expect an rvalue that will cause a function-to-pointer standard conversion?

For an ordinary function call, the postfix expression shall be either an lvalue that refers to a function (in which case the function-to-pointer standard conversion (4.3) is suppressed on the postfix expression), or it shall have pointer to function type. 5.2.2p1

The suppression of the function-to-pointer conversion is a difference in specification from C, but the final behavior is the same.

If either the second or the third operand has type (possibly cv-qualified) **`void`***, then the . . . , and function-to-pointer (4.3) standard conversions are performed on the second and third operands,* 5.16p2

If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand. 5.17p3

After determining the type of each parameter, any parameter of type "array of `T`*" or "function returning* `T`*" is adjusted to be "pointer to* `T`*" or "pointer to function returning* `T`*," respectively.* 8.3.5p3

This appears to cover all cases.

---

734 53) The name "lvalue" comes originally from the assignment expression **`E1 = E2`**, in which the left operand **`E1`** is required to be a (modifiable) lvalue. footnote 53

C++

The C++ Standard does not provide a rationale for the origin of the term *lvalue*.

---

736 What is sometimes called "rvalue" is in this International Standard described as the "value of an expression". rvalue

C++

The C++ Standard uses the term *rvalue* extensively, but the origin of the term is never explained.

3.10p1

Every expression is either an lvalue or an rvalue.

footnote 54

54) Because this conversion does not occur, the operand of the **sizeof** operator remains a function designator and violates the constraint in 6.5.3.4. 739

C++

The C++ Standard does not specify that use of such a usage renders a program ill-formed:

5.3.3p1 *The **sizeof** operator shall not be applied to an expression that has function or incomplete type, or to an enumeration type before all its enumerators have been declared, or to the parenthesized name of such types, or to an lvalue that designates a bit-field.*

6.3.2.2 void

void expression

The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression. 740

C++

3.9.1p9 *An expression of type **void** shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), as a second or third operand of **?:** (5.16), as the operand of **typeid**, or as the expression in a return statement (6.6.3) for a function with the return type **void**.*

The C++ Standard explicitly enumerates the contexts in which a void expression can appear. The effect is to disallow the value of a void expression being used, or explicitly converted, as per the C wording.
The C++ Standard explicitly permits the use of a void expression in a context that is not supported in C:

```
extern void g(void);

void f(void)
{
return g(); /* Constraint violation. */
}
```

5.2.9p4 *Any expression can be explicitly converted to type "cv **void**."*

Thus, C++ supports the **void** casts allowed in C.

If an expression of any other type is evaluated as a void expression, its value or designator is discarded. 741

C90

If an expression of any other type occurs in a context where a void expression is required, its value or designator is discarded.

The wording in the C90 Standard begs the question, "When is a void expression required"?

C++

There are a number of contexts in which an expression is evaluated as a void expression in C. In two of these cases the C++ Standard specifies that lvalue-to-rvalue conversions are not applied: Clauses 5.18p1 left operand of the comma operator, and 6.2p1 the expression in an expression statement. The other context is an explicit cast:

> *Any expression can be explicitly converted to type "`cv` **`void`**." The expression value is discarded.* 5.2.9p4

So in C++ there is no value to discard in these contexts. No other standards wording is required.

---

742 (A void expression is evaluated for its side effects.)

C++

This observation is not made in the C++ Standard.

6.3.2.3 Pointers

---

743 A pointer to **`void`** may be converted to or from a pointer to any incomplete or object type. pointer to void converted to/from

C++

> *An rvalue of type "pointer to `cv` **`void`**" can be explicitly converted to a pointer to object type.* 5.2.9p10

In C++ incomplete types, other than *cv* **void**, are included in the set of object types. 475 object types
In C++ the conversion has to be explicit, while in C it can be implicit. C source code that relies on an implicit conversion being performed by a translator will not be accepted by a C++ translator.

The suggested resolution to SC22/WG21 DR #137 proposes changing the above sentence, from 5.2.9p10, to:

> *An rvalue of type "pointer to `cv1` **`void`**" can be converted to an rvalue of type "pointer to `cv2 >T`", where `T` is an object type and `cv2` is the same cv-qualification as, or greater cv-qualification than, `cv1`.* Proposed change to C++

If this proposal is adopted, a pointer-to qualified type will no longer, in C++, be implicitly converted unless the destination type is at least as well qualified.

---

744 A pointer to any incomplete or object type may be converted to a pointer to **`void`** and back again; pointer converted to pointer to void

C++

> *Except that converting an rvalue of type "pointer to `T1`" to the type "pointer to `T2`" (where `T1` and `T2` are object types and where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.* 5.2.10p7

The C++ wording is more general than that for C. A pointer can be converted to any pointer type and back again, delivering the original value, provided the relative alignments are no stricter.

Source developed using a C++ translator may make use of pointer conversion sequences that are not required to be supported by a C translator.

---

745 the result shall compare equal to the original pointer. converted via pointer to void compare equal

C++

5.2.9p10 *A value of type pointer to object converted to "pointer to* `cv` **`void`***" and back to the original pointer type will have its original value.*

In C++ incomplete types, other than *cv* **`void`**, are included in the set of object types.

---

pointer converting qualified/unqualified

For any qualifier *q*, a pointer to a non-*q*-qualified type may be converted to a pointer to the *q*-qualified version of the type; 746

C++

4.4p1 *An rvalue of type "pointer to* `cv1` *T" can be converted to an rvalue of type "pointer to* `cv2` *T" if "*`cv2` *T" is more cv-qualified than "*`cv1` *T."*

4.4p2 *An rvalue of type "pointer to member of X of type* `cv1` *T" can be converted to an rvalue of type "pointer to member of X of type* `cv2` *T" if "*`cv2` *T" is more cv-qualified than "*`cv1` *T."*

---

qualified/unqualified pointer compare equal

the values stored in the original and converted pointers shall compare equal. 747

C++

3.9.2p3 *Pointers to cv-qualified and cv-unqualified versions (3.9.3) of layout-compatible types shall have the same value representation and alignment requirements (3.9).*

By specifying layout-compatible types, not the same type, the C++ Standard restricts the freedom of implementations more than C99 does.

---

null pointer constant

An integer constant expression with the value 0, or such an expression cast to type **`void *`**, is called a *null pointer constant*.[55)] 748

C++

4.10p1 *A null pointer constant is an integral constant expression (5.19) rvalue of integer type that evaluates to zero.*

The C++ Standard only supports the use of an integer constant expression with value `0`, as a null pointer constant. A program that explicitly uses the pointer cast form need not be conforming C++; it depends on the context in which it occurs. Use of the implementation-provided `NULL` macro avoids this compatibility problem by leaving it up to the implementation to use the appropriate value.

The C++ Standard specifies the restriction that a null pointer constant can only be created at translation time.

Footnote 64

64) Converting an integral constant expression (5.19) with value zero always yields a null pointer (4.10), but converting other expressions that happen to have value zero need not yield a null pointer.

---

749 If a null pointer constant is converted to a pointer type, the resulting pointer, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function. null pointer

C++

A null pointer constant can be converted to a pointer type; the result is the null pointer value of that type and is distinguishable from every other value of pointer to object or pointer to function type. 4.10p1

A null pointer constant (4.10) can be converted to a pointer to member type; the result is the null member pointer value of that type and is distinguishable from any pointer to member not created from a null pointer constant. 4.11p1

Presumably *distinguishable* means that the pointers will compare unequal.

Two pointers of the same type compare equal if and only if they are both null, both point to the same object or function, or both point one past the end of the same array. 5.10p1

From which we can deduce that a null pointer constant cannot point one past the end of an object either.

---

750 Conversion of a null pointer to another pointer type yields a null pointer of that type. null pointer conversion yields null pointer

C90

The C90 Standard was reworded to clarify the intent by the response to DR #158.

---

751 Any two null pointers shall compare equal. null pointer compare equal

C++

Two null pointer values of the same type shall compare equal. 4.10p1

Two null member pointer values of the same type shall compare equal. 4.11p1

The C wording does not restrict the null pointers from being the same type.

The null pointer value is converted to the null pointer value of the destination type. 4.10p3

This handles pointers to class type. The other cases are handled in 5.2.7p4, 5.2.9p8, 5.2.10p8, and 5.2.11p6.

---

753 Except as previously specified, the result is implementation-defined, might not be correctly aligned, might not point to an entity of the referenced type, and might be a trap representation.[56] integer-to-pointer implementation-defined

C++

The C++ Standard specifies the following behavior for the **`reinterpret_cast`**, which is equivalent to the C cast operator in many contexts.

5.2.10p5 *A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined.*

The C++ Standard provides a guarantee— a round path conversion via an integer type of sufficient size (provided one exists) delivers the original value. Source developed using a C++ translator may contain constructs whose behavior is implementation-defined in C.

The C++ Standard does not discuss trap representations for anything other than floating-point types.

---

pointer permission to convert to integer

Any pointer type may be converted to an integer type. 754

C++

5.2.10p4 *A pointer can be explicitly converted to any integral type large enough to hold it.*

The C++ wording is more restrictive than C, which has no requirement that the integer type be large enough to hold the pointer.

integer-to-pointer implementation-defined 753

While the specification of the conversion behaviors differ between C++ and C (undefined vs. implementation-defined, respectively), differences in the processor architecture is likely to play a larger role in the value of the converted result.

---

pointer conversion undefined behavior

If the result cannot be represented in the integer type, the behavior is undefined. 756

C90

If the space provided is not long enough, the behavior is undefined.

The C99 specification has moved away from basing the specification on storage to a more general one based on representation.

C++

The C++ Standard does not explicitly specify any behavior when the result cannot be represented in the integer type. (The wording in 5.2.10p4 applies to "any integral type large enough to hold it.")

The result need not be in the range of values of any integer type. 757

C90

The C90 requirement was based on sufficient bits being available, not representable ranges.

C++

There is no equivalent permission given in the C++ Standard.

---

pointer converted to pointer to different object or type

A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. 758

C++

The C++ Standard states this in 5.2.9p5, 5.2.9p8, 5.2.10p7 (where the wording is very similar to the C wording), and 5.2.11p10.

759 If the resulting pointer is not correctly aligned[57)] for the pointed-to type, the behavior is undefined.

C++

Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified. 5.2.10p7

The unspecified behavior occurs if the pointer is not cast back to its original type, or the relative alignments are stricter.

Source developed using a C++ translator may contain a conversion of a pointer value that makes use of unspecified behavior, but causes undefined behavior when processed by a C translator.

760 Otherwise, when converted back again, the result shall compare equal to the original pointer. pointer converted back to pointer

C++

Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified. 5.2.10p7

The C++ Standard does not specify what *original pointer value* means (e.g., it could be interpreted as bit-for-bit equality, or simply that the two values compare equal).

761 When a pointer to an object is converted to a pointer to a character type, the result points to the lowest addressed byte of the object. pointer converted to pointer to character object lowest addressed byte

C90

The C90 Standard does not explicitly specify this requirement.

C++

*The result of converting a "pointer to `cv T`" to a "pointer to `cv` **`void`**" points to the start of the storage location where the object of type `T` resides, . . .* 4.10p2

However, the C wording is for pointer-to character type, not pointer to **`void`**.

A cv-qualified or cv-unqualified (3.9.3) `void*` shall have the same representation and alignment requirements as a cv-qualified or cv-unqualified `char*`. 3.9.2p4

A pointer to an object can be explicitly converted to a pointer to an object of different type[65)]. Except that converting an rvalue of type "pointer to T1" to the type "pointer to T2" (where T1 and T2 are object types and where the alignment requirements of T2 are no stricter than those of T1) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified. 5.2.10p7

The C++ Standard does not require the result of the conversion to be a pointer to the lowest addressed byte of the object. However, it is very likely that C++ implementations will meet the C requirement.

footnote 56 pointer/integer consistent mapping

56) The mapping functions for converting a pointer to an integer or an integer to a pointer are intended to be consistent with the addressing structure of the execution environment. 763

C++

5.2.10p4 *[Note: it is intended to be unsurprising to those who know the addressing structure of the underlying machine.]*

Is an unsurprising mapping the same as a consistent one? Perhaps an unsurprising mapping is what C does. :-)

footnote 57

57) In general, the concept "correctly aligned" is transitive: if a pointer to type A is correctly aligned for a pointer to type B, which in turn is correctly aligned for a pointer to type C, then a pointer to type A is correctly aligned for a pointer to type C. 764

C90

This observation was not explicitly specified in the C90 Standard.

C++

The C++ Standard does not point out that alignment is a transitive property.

object point at each bytes of

Successive increments of the result, up to the size of the object, yield pointers to the remaining bytes of the object. 765

C90

The C90 Standard does not explicitly specify this requirement.

C++

The equivalent C++ requirement is only guaranteed to apply to pointer to **void** and it is not possible to perform arithmetic on this pointer type. However, in practice C++ implementations are likely to meet this C requirement.

call function via converted pointer

If a converted pointer is used to call a function whose type is not compatible with the pointed-to type, the behavior is undefined. 768

C++

5.2.10p6 *The effect of calling a function through a pointer to a function type (8.3.5) that is not the same as the type used in the definition of the function is undefined.*

compatible type if 631

C++ requires the parameter and return types to be the same, while C only requires that these types be compatible. However, the only difference occurs when an enumerated type and its compatible integer type are intermixed.

6.4 Lexical elements

token syntax preprocessing token syntax

770

```
token:
        keyword
        identifier
        constant
        string-literal
```

```
                punctuator
preprocessing-token:
                header-name

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

C90

The *non-terminal* operator was included as both a *token* and *preprocessing-token* in the C90 Standard. Tokens that were operators in C90 have been added to the list of punctuators in C99.

C++

C++ maintains the C90 distinction between operators and punctuators. C++ also classifies what C calls a constant as a literal, a string-literal as a literal and a C character-constant is known as a character-literal.

Constraints

Semantics

---

772 A *token* is the minimal lexical element of the language in translation phases 7 and 8.

C++

The C++ Standard makes no such observation.

---

773 The categories of tokens are: keywords, identifiers, constants, string literals, and punctuators.

C90

Tokens that were defined to be operators in C90 have been added to the list of punctuators in C99.

C++

> *There are five kinds of tokens: identifiers, keywords, literals,[18] operators, and other separators.* 2.6p1

What C calls constants, C++ calls literals. What C calls punctuators, C++ breaks down into operators and punctuators.

---

775 The categories of preprocessing tokens are: header names, identifiers, preprocessing numbers, character constants, string literals, punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories.[58)]

C++

In clause 2.4p2, apart from changes to the terminology, the wording is identical.

footnote 58

58) An additional category, placemarkers, is used internally in translation phase 4 (see 6.10.3.3); it cannot occur in source files. 781

C90

The term *placemarker* is new in C99. They are needed to describe the behavior when an empty macro argument is the operand of the **##** operator, which could not occur in C90.

C++

This category was added in C99 and does not appear in the C++ Standard, which has specified the preprocessor behavior by copying the words from C90 (with a few changes) rather than providing a reference to the C Standard.

header name exception to rule

There is one exception to this rule: ~~a header name preprocessing token is only recognized within a **#include** preprocessing directive, and within such a directive,~~ Header name preprocessing tokens are recognized only within **#include** preprocessing directives or in implementation-defined locations within **#pragma** directives. 783

C90

This exception was not called out in the C90 Standard and was added by the response to DR #017q39.

C++

This exception was not called out in C90 and neither is it called out in the C++ Standard.

6.4.1 Keywords

keyword: one of 788

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

C90

Support for the keywords **restrict**, **_Bool**, **_Complex**, and **_Imaginary** is new in C99.

C++

The C++ Standard includes the additional keywords:

bool	**mutable**	**this**
catch	**namespace**	**throw**
class	**new**	**true**
const_cast	**operator**	**try**
delete	**private**	**typeid**
dynamic_cast	**protected**	**typename**
explicit	**public**	**using**
export	**reinterpret_cast**	**virtual**
false	**static_cast**	**wchar_t**
friend	**template**	

The C++ Standard does not include the keywords **restrict**, **_Bool**, **_Complex**, and **_Imaginary**. However, identifiers beginning with an underscore followed by an uppercase letter is reserved for use by C++ implementations (17.4.3.1.2p1). So, three of these keywords are not available for use by developers.

In C the identifier `wchar_t` is a typedef name defined in a number of headers; it is not a keyword.

The C99 header **`<stdbool.h>`** defines macros named `bool`, `true`, `false`. This header is new in C99 and is not one of the ones listed in the C++ Standard as being supported by that language.

Semantics

6.4.2 Identifiers

6.4.2.1 General

---

792 identifier syntax

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

C90

Support for *universal-character-name* and "other implementation-defined characters" is new in C99.

C++

The C++ Standard uses the term *nondigit* to denote an *identifier-nondigit*. The C++ Standard does not specify the use of *other implementation-defined characters*. This is because such characters will have been replaced in translation phase 1 and not be visible here. 116 translation phase 1

Semantics

---

793 An identifier is a sequence of nondigit characters (including the underscore _, the lowercase and uppercase Latin letters, and other characters) and digits, which designates one or more entities as described in 6.2.1.

C90

Explicit support for other characters is new in C99.

---

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

C90

The C90 Standard does not explicitly state this fact.

---

796 Each universal character name in an identifier shall designate a character whose encoding in ISO/IEC 10646 falls into one of the ranges specified in annex D.[60) identifier UCN

C90

Support for universal character names is new in C99.

---

797 The initial character shall not be a universal character name designating a digit.

C++

This requirement is implied by the terminal non-name used in the C++ syntax. Annex E of the C++ Standard does not list any UCN digits in the list of supported UCN encodings.

---

identifier multibyte character in

An implementation may allow multibyte characters that are not part of the basic source character set to appear in identifiers; 798

C90

This permission is new in C99.

C++

translation phase 1 116

The C++ Standard does not explicitly contain this permission. However, translation phase 1 performs an implementation-defined mapping of the source file characters, and an implementation may choose to support multibyte characters in identifiers via this route.

When preprocessing tokens are converted to tokens during translation phase 7, if a preprocessing token could be converted to either a keyword or an identifier, it is converted to a keyword. 800

C90

This wording is a simplification of the convoluted logic needed in the C90 Standard to deduce from a constraint what C99 now says in semantics. The removal of this C90 constraint is not a change of behavior, since it was not possible to write a program that violated it.

C90 6.1.2

Constraints

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

---

footnote 60

60) On systems in which linkers cannot accept extended characters, an encoding of the universal character name may be used in forming valid external identifiers. 801

C90

extended characters 215

Extended characters were not available in C90, so the suggestion in this footnote does not apply.

Implementation limits

Implementation limits

As discussed in 5.2.4.1, an implementation may limit the number of significant initial characters in an identifier; 804

C90

The C90 Standard does not contain this observation.

C++

2.10p1

All characters are significant.[20)]

C identifiers that differ after the last significant character will cause a diagnostic to be generated by a C++ translator.

Annex B contains an informative list of possible implementation limits. However, “ . . . these quantities are only guidelines and do not determine compliance.”.

The number of significant characters in an identifier is implementation-defined. 806

C++

> *All characters are significant.*[20] 2.10p1

References to the same C identifier, which differs after the last significant character, will cause a diagnostic to be generated by a C++ translator.

There is also an informative annex which states:

> *Number of initial characters in an internal identifier or a macro name [1024]* Annex Bp2
>
> *Number of initial characters in an external identifier [1024]*

---

808 If two identifiers differ only in nonsignificant characters, the behavior is undefined.

C++

In C++ all characters are significant, thus this statement does not apply in C++.

6.4.2.2 Predefined identifiers

Semantics

---

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

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

appeared, where *function-name* is the name of the lexically-enclosing function.[61)]

C90

Support for the identifier **__func__** is new in C99.

C++

Support for the identifier **__func__** is new in C99 and is not available in the C++ Standard.

---

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

C90

Names beginning with two underscores were specified as reserved for any use by the C90 Standard. The following program is likely to behave differently when translated and executed by a C99 implementation.

```
#include <stdio.h>

int main(void)
{
int __func__ = 1;

printf("d\n", __func__);
}
```

C++

Names beginning with __ are reserved for use by a C++ implementation. This leaves the way open for a C++ implementation to use this name for some purpose.

6.4.3 Universal character names

universal character name syntax

815

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

C90

Support for this syntactic category is new in C99.

Constraints

UCNs not basic character set

A universal character name shall not specify a character whose short identifier is less than 00A0 other than 0024 (`$`), 0040 (`@`), or 0060 (`‘`), nor one in the range D800 through DFFF inclusive.[62)] 816

C++

2.2p2 *If the hexadecimal value for a universal character name is less than 0x20 or in the range 0x7F–0x9F (inclusive), or if the universal character name designates a character in the basic source character set, then the program is ill-formed.*

The range of hexadecimal values that are not permitted in C++ is a subset of those that are not permitted in C. This means that source which has been accepted by a conforming C translator will also be accepted by a conforming C++ translator, but not the other way around.

Description

Universal character names may be used in identifiers, character constants, and string literals to designate characters that are not in the basic character set. 817

C++

The C++ Standard also supports the use of universal character names in these contexts, but does not say in words what it specifies in the syntax (although 2.2p2 comes close for identifiers).

Semantics

footnote 62

62) The disallowed characters are the characters in the basic character set and the code positions reserved by ISO/IEC 10646 for control characters, the character DELETE, and the S-zone (reserved for use by UTF-16). 820

C++

The C++ Standard does not make this observation.

6.4.4 Constants

822 constant syntax

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

C++

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

The C++ Standard also includes `string-literal` and `boolean-literal` in the list of literals, but it does not include enumeration constants in the list of literals. However:

The identifiers in an `enumerator-list` *are declared as constants, and can appear wherever constants are required.* 7.2p1

The C++ terminology more closely follows common developer terminology by using *literal* (a single token) and *constant* (a sequence of operators and literals whose value can be evaluated at translation time). The value of a literal is explicit in the sequence of characters making up its token. A constant may be made up of more than one token or be an identifier. The operands in a constant have to be evaluated by the translator to obtain its result value. C uses the more easily confused terminology of `integer-constant` (a single token) and `constant-expression` (a sequence of operators, `integer-constant` and `floating-constant` whose value can be evaluated at translation time).

Constraints

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

C++

The C++ Standard has equivalent wording covering `integer-literals` (2.13.1p3) `character-literals` (2.13.2p3) and `floating-literals` (2.13.3p1). For `enumeration-literals` their type depends on the context in which the question is asked:

Following the closing brace of an `enum-specifier`*, each enumerator has the type of its enumeration. Prior to the closing brace, the type of each enumerator is the type of its initializing value.* 7.2p4

7.2p5

The underlying type of an enumeration is an integral type that can represent all the enumerator values defined in the enumeration.

Semantics

constant type determined by form and value

Each constant ~~has a type, determined by its form and value, as detailed later.~~ shall have a type and the value of a constant shall be in the range of representable values for its type. 824

C++

2.13.1p2 *The type of an integer literal depends on its form, value, and suffix.*

2.13.3p1 *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`**.

There are no similar statements for the other kinds of literals, although C++ does support suffixes on the floating types. However, the syntactic form of string literals, character literals, and boolean literals determines their type.

6.4.4.1 Integer constants

integer constant syntax

825

```
integer-constant:
        decimal-constant integer-suffix_opt
        octal-constant integer-suffix_opt
        hexadecimal-constant integer-suffix_opt
decimal-constant:
        nonzero-digit
        decimal-constant digit
octal-constant:
        0
        octal-constant octal-digit
hexadecimal-constant:
        hexadecimal-prefix hexadecimal-digit
        hexadecimal-constant hexadecimal-digit
hexadecimal-prefix: one of
        0x  0X
nonzero-digit: one of
        1  2  3  4  5  6  7  8  9
octal-digit: one of
        0  1  2  3  4  5  6  7
hexadecimal-digit: one of
        0  1  2  3  4  5  6  7  8  9
        a  b  c  d  e  f
        A  B  C  D  E  F
integer-suffix:
        unsigned-suffix long-suffix_opt
        unsigned-suffix long-long-suffix
        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

C90

Support for *long-long-suffix* and the nonterminal *hexadecimal-prefix* is new in C99.

C++

The C++ syntax is identical to the C90 syntax.
Support for *long-long-suffix* and the nonterminal *hexadecimal-prefix* is not available in C++.

Description

Semantics

836 integer constant possible types

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

C90

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

Support for the type **long long** is new in C99.

The C90 Standard will give a sufficiently large decimal constant, which does not contain a *u* or *U* suffix— the type **unsigned long**. The C99 Standard will never give a decimal constant that does not contain either of these suffixes— an unsigned type.

Because of the behavior of C++, the sequencing of some types on this list has changed from C90. The following shows the entries for the C90 Standard that have changed.

Suffix	Decimal Constant
none	**int** **long int** **unsigned long int**
l or **L**	**long int** **unsigned long int**

Under C99, the none suffix, and *l* or *L* suffix, case no longer contain an unsigned type on their list.

A decimal constant, unless given a *u* or *U* suffix, is always treated as a signed type.

C++

2.13.1p2 *If it is decimal and has no suffix, it has the first of these types in which its value can be represented:* ***int***, ***long int***; *if the value cannot be represented as a* ***long int***, *the behavior is undefined. If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented:* ***int***, ***unsigned int***, ***long int***, ***unsigned long int***. *If it is suffixed by* u *or* U, *its type is the first of these types in which its value can be represented:* ***unsigned int***, ***unsigned long int***. *If it is suffixed by* l *or* L, *its type is the first of these types in which its value can be represented:* ***long int***, ***unsigned long int***. *If it is suffixed by* ul, lu, uL, Lu, Ul, lU, UL, *or* LU, *its type is* ***unsigned long int***.

The C++ Standard follows the C99 convention of maintaining a decimal constant as a signed and never an unsigned type.

The type **long long**, and its unsigned partner, is not available in C++.

There is a difference between C90 and C++ in that the C90 Standard can give a sufficiently large decimal literal that does not contain a *u* or *U* suffix— the type **unsigned long**. Neither the C++ or C99 Standard will give a decimal constant that does not contain either of these suffixes— an unsigned type.

If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. 837

C90

Explicit support for extended types is new in C99.

C++

The C++ Standard allows new object types to be created. It does not specify any mechanism for giving literals these types.

A C translation unit that contains an integer constant that has an extended integer type may not be accepted by a conforming C++ translator. But then it may not be accepted by another conforming C translator either. Support for the construct is implementation-defined.

6.4.4.2 Floating constants

floating constant syntax

842

floating-constant:

```
        decimal-floating-constant
        hexadecimal-floating-constant
decimal-floating-constant:
        fractional-constant exponent-part_opt floating-suffix_opt
        digit-sequence exponent-part floating-suffix_opt
hexadecimal-floating-constant:
        hexadecimal-prefix hexadecimal-fractional-constant
                binary-exponent-part floating-suffix_opt
        hexadecimal-prefix hexadecimal-digit-sequence
                binary-exponent-part floating-suffix_opt
fractional-constant:
        digit-sequence_opt . digit-sequence
        digit-sequence .
exponent-part:
        e sign_opt digit-sequence
        E sign_opt digit-sequence
sign: one of
        + -
digit-sequence:
        digit
        digit-sequence digit
hexadecimal-fractional-constant:
        hexadecimal-digit-sequence_opt .
                hexadecimal-digit-sequence
        hexadecimal-digit-sequence .
binary-exponent-part:
        p sign_opt digit-sequence
        P sign_opt digit-sequence
hexadecimal-digit-sequence:
        hexadecimal-digit
        hexadecimal-digit-sequence hexadecimal-digit
floating-suffix: one of
        f l F L
```

C90

Support for *hexadecimal-floating-constant* is new in C99. The terminal *decimal-floating-constant* is new in C99 and its right-hand side appeared on the right of *floating-constant* in the C90 Standard.

C++

The C++ syntax is identical to that given in the C90 Standard.
Support for *hexadecimal-floating-constant* is not available in C++.

Description

844 The components of the significand part may include a digit sequence representing the whole-number part, followed by a period (.), followed by a digit sequence representing the fraction part. whole-number part fraction part

C++

The integer part, the optional decimal point and the optional fraction part form the significant part of the floating literal.

The use of the term *significant* may be a typo. This term does not appear in the C++ Standard and it is only used in this context in one paragraph.

The components of the exponent part are an **e**, **E**, **p**, or **P** followed by an exponent consisting of an optionally signed digit sequence. 845

C90

Support for *p* and *P* is new in C99.

C++

Like C90, the C++ Standard does not support the use of *p*, or *P*.

Semantics

The significand part is interpreted as a (decimal or hexadecimal) rational number; 848

C90

Support for hexadecimal significands is new in C99.

C++

The C++ Standard does not support hexadecimal significands, which are new in C99.

---

the digit sequence in the exponent part is interpreted as a decimal integer. 849

C++

2.13.3p1 *. . . , an optionally signed integer exponent, . . .*

There is no requirement that this integer exponent be interpreted as a decimal integer. Although there is wording specifying that both the integer and fraction parts are in base 10, there is no such wording for the exponent part. It would be surprising if the C++ Standard were to interpret 1.2e011 as representing 1.2×10^9; therefore this issue is not specified as a difference.

For hexadecimal floating constants, the exponent indicates the power of 2 by which the significand part is to be scaled. 851

C90

Support for hexadecimal floating constants is new in C99.

C++

The C++ Standard does not support hexadecimal floating constants.

---

floating constant representable value chosen

For decimal floating constants, and also for hexadecimal floating constants when **FLT_RADIX** is not a power of 2, the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner. 852

C90

Support for hexadecimal floating constants is new in C99.

C++

2.13.3p1

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.

---

857 Floating constants are converted to internal format as if at translation-time. floating constant internal format

C90

No such requirement is explicitly specified in the C90 Standard.

In C99 floating constants may convert to more range and precision than is indicated by their type; that is, `0.1f` may be represented as if it had been written `0.1L`. 354 FLT_EVAL_METHOD

C++

Like C90, there is no such requirement in the C++ Standard.

---

858 The conversion of a floating constant shall not raise an exceptional condition or a floating-point exception at execution time. floating constant conversion not raise exception

C90

No such requirement was explicitly specified in the C90 Standard.

C++

Like C90, there is no such requirement in the C++ Standard.

Recommended practice

---

859 The implementation should produce a diagnostic message if a hexadecimal constant cannot be represented exactly in its evaluation format; hexadecimal constant not represented exactly

C90

Recommended practices are new in C99, as are hexadecimal floating constants.

C++

The C++ Standard does not specify support for hexadecimal floating constants.

---

861 The translation-time conversion of floating constants should match the execution-time conversion of character strings by library functions, such as **`strtod`**, given matching inputs suitable for both conversions, the same result format, and default execution-time rounding.[64)]

C90

This recommendation is new in C99.

C++

No such requirement is explicitly specified in the C++ Standard.

---

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

C++

There observation is not made in the C++ Standard. The C++ Standard includes the C library by reference, so by implication this statement is also true in C++.

6.4.4.3 Enumeration constants

enumeration-constant:
identifier 863

C++

The C++ syntax uses the terminator *enumerator*.

Semantics

enumeration constant type

An identifier declared as an enumeration constant has type **int**. 864

C++

7.2p4 *Following the closing brace of an enum-specifier, each enumerator has the type of its enumeration. Prior to the closing brace, the type of each enumerator is the type of its initializing value. If an initializer is specified for an enumerator, the initializing value has the same type as the expression. If no initializer is specified for the first enumerator, the type is an unspecified integral type. Otherwise the type is the same as the type of the initializing value of the preceding enumerator unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value.*

This is quite a complex set of rules. The most interesting consequence of them is that each enumerator, in the same type definition, can have a different type (at least while the enumeration is being defined):
In C the type of the enumerator is always **int**. In C++ it can vary, on an enumerator by enumerator basis, for the same type definition. This behavior difference is only visible outside of the definition if an initializing value is calculated by applying the **sizeof** operator to a prior enumerator in the current definition.

```
#include <limits.h>

enum TAG { E1 = 2L,             // E1 has type long
           E2 = sizeof(E1),     // E2 has type size_t, value sizeof(long)
           E3 = 9,              // E3 has type int
           E4 = '4',            // E4 has type char
           E5 = INT_MAX,        // E5 has type int
           E6,                  // is E6 an unsigned int, or a long?
           E7 = sizeof(E4),     // E2 has type size_t, value sizeof(char)
         }                      // final type is decided when the } is encountered
           e_val;

int probably_a_C_translator(void)
{
return (E2 == E7);
}
```

Source developed using a C++ translator may contain enumeration with values that would cause a constraint violation if processed by a C translator.

```
#include <limits.h>

enum TAG { E1 = LONG_MAX }; /* Constraint violation if LONG_MAX != INT_MAX */
```

6.4.4.4 Character constants

866 character constant syntax escape sequence syntax

```
character-constant:
                ' c-char-sequence '
                L' c-char-sequence '

c-char-sequence:
                c-char
                c-char-sequence c-char
c-char:
                any member of the source character set except
                       the single-quote ', backslash \, or new-line character
                escape-sequence
escape-sequence:
                simple-escape-sequence
                octal-escape-sequence
                hexadecimal-escape-sequence
                universal-character-name
simple-escape-sequence: one of
                \'  \"  \?  \\
                \a  \b  \f  \n  \r  \t  \v
octal-escape-sequence:
                \ octal-digit
                \ octal-digit octal-digit
                \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
                \x hexadecimal-digit
                hexadecimal-escape-sequence hexadecimal-digit
```

C90

Support for *universal-character-name* is new in C99.

C++

The C++ Standard classifies *universal-character-name* as an *escape-sequence*, not as a *c-char*. This makes no difference in practice to the handling of such *c-char*s.

Description

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

C90

The example of ab as an integer character constant has been removed from the C99 description.

C++

> *A character literal is one or more characters enclosed in single quotes, as in* 'x', . . . 2.13.2p1

A multibyte character is replaced by a *universal-character-name* in C++ translation phase 1. So, the C++ Standard does not need to refer to such entities here.

With a few exceptions detailed later, the elements of the sequence are any members of the source character set; 869

C++

2.13.2p5 *[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-name`*s. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained.]*

In C++ all elements in the sequence are characters in the source character set after translation phase 1. The creation of *`character-literal`* preprocessing tokens occurs in translation phase 3, rendering this statement not applicable to C++.

character constant mapped

they are mapped in an implementation-defined manner to members of the execution character set. 870

C++

2.13.2p1 *An ordinary character literal that contains a single* `c-char` *has type* **`char`***, with value equal to the numerical value of the encoding of the* `c-char` *in the execution character set.*

2.2p3 *The values of the members of the execution character sets are implementation-defined, . . .*

2.13.2p2 *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.*

Taken together, these statements have the same meaning as the C specification.

escape sequences character constant character constant escape sequences

The single-quote `'`, the double-quote `"`, the question-mark `?`, the backslash `\`, and arbitrary integer values are representable according to the following table of escape sequences: 871

```
single quote   '       \'
double quote   "       \"
question mark  ?       \?
backslash      \       \\
octal character        \octal digits
hexadecimal character  \xhexadecimal digits
```

C++

The C++ wording, in Clause 2.13.2p3, does discuss arbitrary integer values and the associated Table 5 includes all of the defined escape sequences.

In addition, characters not in the basic character set are representable by universal character names and certain nongraphic characters are representable by escape sequences consisting of the backslash `\` followed by a lowercase letter: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, and `\v`.[65] 878

C90

Support for universal character names is new in C99.

C++

Apart from the rule of syntax given in Clause 2.13.2 and Table 5, there is no other discussion of these escape sequences in the C++ Standard. :-O

---

880 If any other character follows a backslash, the result is not a token and a diagnostic is required.

C90

If any other escape sequence is encountered, the behavior is undefined.

C++

There is no equivalent sentence in the C++ Standard. However, this footnote is intended to explicitly spell out what the C syntax specifies. The C++ syntax specification is identical, but the implications have not been explicitly called out.

Constraints

---

882 The value of an octal or hexadecimal escape sequence shall be in the range of representable values for the type **unsigned char** for an integer character constant, or the unsigned type corresponding to **wchar_t** for a wide character constant. escape sequence value within range

C++

The value of a character literal is implementation-defined if it falls outside of the implementation-defined range defined for char (for ordinary literals) or **wchar_t** *(for wide literals).* 2.13.2p4

The wording in the C++ Standard applies to the entire character literal, not to just a single character within it (the C case). In practice this makes no difference because C++ does not provide the option available to C implementations of allowing more than one character in an integer character constant.

The range of values that can be represented in the type **char** may be a subset of those representable in the type **unsigned char**. In some cases defined behavior in C becomes implementation-defined behavior in C++.

```
char *p = "\0x80"; /* does not affect the conformance status of the program */
                   // if CHAR_MAX is 127, behavior is implementation-defined
```

In C a value outside of the representable range causes a diagnostic to be issued. The C++ behavior is implementation-defined in this case. Source developed using a C++ translator may need to be modified before it is acceptable to a C translator.

Semantics

---

883 An integer character constant has type **int**. character constant type

C++

2.13.2p1

> *An ordinary character literal that contains a single* `c-char` *has type* **char**, . . .

The only visible effect of this difference in type, from the C point of view, is the value returned by sizeof. In the C++ case the value is always 1, while in C the value is the same as sizeof(int), which could have the value 1 (for some DSP chips), but for most implementations is greater than 1.

2.13.2p1

> *A multicharacter literal has type* **int** *and implementation-defined value.*

The behavior in this case is identical to C.

---

character constant more than one character

The value of an integer character constant containing more than one character (e.g., **'ab'**), or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined. 885

C90

> *The value of an integer character constant containing more than one character, or containing a character or escape sequence not represented in the basic execution character set, is implementation-defined.*

C++

889 wide character escape sequence implementation-defined

The C++ Standard does not include any statement covering escape sequences that are not represented in the execution character set. The other C requirements are covered by words (2.13.2p1) having the same meaning.

---

character constant single character value

If an integer character constant contains a single character or escape sequence, its value is the one that results when an object with type **char** whose value is that of the single character or escape sequence is converted to type **int**. 886

C++

The requirement contained in this sentence is not applicable to C++ because this language gives character literals the type **char**. There is no implied conversion to **int** in C++.

---

wide character constant type of

A wide character constant has type **wchar_t**, an integer type defined in the **<stddef.h>** header. 887

C++

2.13.2p2

> *A wide-character literal has type* **wchar_t**.[23)]

In C++ **wchar_t** is one of the basic types (it is also a keyword). There is no need to define it in the **<stddef.h>** header.

3.9.1p5

> *Type* **wchar_t** *shall have the same size, signedness, and alignment requirements (3.9) as one of the other integral types, called its underlying type.*

Although C++ includes the C library by reference, the **<stddef.h>** header, in a C++ implementation, cannot contain a definition of the type **wchar_t**, because **wchar_t** is a keyword in C++. It is thus possible to use the type **wchar_t** in C++ without including the **<stddef.h>** header.

---

multibyte character mapped by mbtowc

The value of a wide character constant containing a single multibyte character that maps to a member of the extended execution character set is the wide character corresponding to that multibyte character, as defined by the **mbtowc** function, with an implementation-defined current locale. 888

C++

> *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.* 2.13.2p2

The C++ Standard includes the `mbtowc` function by including the C90 library by reference. However, it does not contain any requirement on the values of wide character literals corresponding to the definitions given for the `mbtowc` function (and its associated locale).

There is no requirement for C++ implementations to use a wide character mapping corresponding to that used by the `mbtowc` library function. However, it is likely that implementations of the two languages, in a given environment, will share the same library.

---

889 The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set, is implementation-defined. wide character escape sequence implementation-defined

C++

The C++ Standard (2.13.2p2) does not include any statement covering escape sequences that are not represented in the execution character set.

6.4.5 String literals

---

895 string literal syntax

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

C++

In C++ a *`universal-character-name`* is not considered to be an escape sequence. It therefore appears on the right side of the *`s-char`* rule.

Description

Semantics

---

899 In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character and wide string literal tokens are concatenated into a single multibyte character sequence.

C90

The C90 Standard does not allow character and wide string literals to be mixed in a concatenation:

> *In translation phase 6, the multibyte character sequences specified by any sequence of adjacent character string literal tokens, or adjacent wide string literal tokens, are concatenated into a single multibyte character sequence.*

The C90 Standard contains the additional sentence:

If a character string literal token is adjacent to a wide string literal token, the behavior is undefined.

C90 does not support the concatenation of a character string literal with a wide string literal.

C++

2.13.4p3 *In translation phase 6 (2.1), adjacent narrow string literals are concatenated and adjacent wide string literals are concatenated. If a narrow string literal token is adjacent to a wide string literal token, the behavior is undefined.*

The C++ specification has the same meaning as that in the C90 Standard. If string literals and wide string literals are adjacent, the behavior is undefined. This is not to say that a translator will not concatenate them, only that such behavior is not guaranteed.

---

string literal type

For character string literals, the array elements have type **`char`**, and are initialized with the individual bytes of the multibyte character sequence; 904

C++

2.13.4p1 *An ordinary string literal has type "array of n* `const char`*" and static storage duration (3.7), where n is the size of the string. . . .*

```
char *g_p = "abc"; /* const applies to the array, not the pointed-to type. */

void f(void)
{
"xyz"@lsquare[]1@rsquare[] = 'Y'; /* relies on undefined behavior, need not be diagnosed */
"xyz"@lsquare[]1@rsquare[] = 'Y'; // ill-formed, object not modifiable lvalue
}
```

---

wide string literal type of

for wide string literals, the array elements have type **`wchar_t`**, and are initialized with the sequence of wide characters corresponding to the multibyte character sequence, as defined by the **`mbstowcs`** function with an implementation-defined current locale. 905

C90

The specification that `mbstowcs` be used as an implementation-defined current locale is new in C99.

C++

2.13.4p1 *An ordinary string literal has type "array of n* `const char`*" and static storage duration (3.7), where n is the size of the string. . . .*

The C++ Standard does not specify that `mbstowcs` be used to define how multibyte characters in a wide string literal be mapped:

2.13.4p5

The size of a wide string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for the terminating `L'\0'`.

The extent to which the C library function `mbstowcs` will agree with the definition given in the C++ Standard will depend on its implementation-defined behavior in the current locale.

---

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

C++

This observation is not made in the C++ document.

---

907 The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set is implementation-defined.

C90

This specification of behavior is new in C99.

C++

Like C90, there is no such explicit specification in the C+ Standard.

---

908 It is unspecified whether these arrays are distinct provided their elements have the appropriate values. string literal distinct array

C++

Clause 2.13.4p4 specifies that the behavior is implementation-defined.

6.4.6 Punctuators

---

912 punctuator syntax

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

C90

Support for **<: :> <% %> %: %:%:** was added in Amendment 1 to the C90 Standard. In the C90 Standard there were separate nonterminals for punctuators and operators. The C99 Standard no longer contains a syntactic category for operators. The two nonterminals are merged in C99, except for **sizeof**, which was listed as an operator in C90.

C++

The C++ nonterminal *preprocessing-op-or-punc* (2.12p1) also includes:

```
::   .*   ->*   new   delete
and   and_eq   bitand   bitor   compl
not   not_eq   or   or_eq   xor   xor_eq
```

The identifiers listed above are defined as macros in the header **<iso646.h>** in C. This header must be included before these identifiers are treated as having their C++ meaning.

Semantics

A punctuator is a symbol that has independent syntactic and semantic significance. 913

C90

A punctuator is a symbol that has independent syntactic and semantic significance but does not specify an operation to be performed that yields a value.

The merging of this distinction between operators and punctuators, in C99, makes no practical difference.

C++

This observation is not made in the C++ Standard.

---

operator Depending on context, it may specify an operation to be performed (which in turn may yield a value or a function designator, produce a side effect, or some combination thereof) in which case it is known as an *operator* (other forms of operator also exist in some contexts). 914

C90

In the C90 Standard operators were defined as a separate syntactic category, some of which shared the same spelling as some punctuators.

An operator specifies an operation to be performed (an evaluation) that yields a value, or yields a designator, or produces a side effect, or a combination thereof.

An *operand* is an entity on which an operator acts. 915

C++

The nearest C++ comes to defining *operand* is:

5p1 *An expression is a sequence of operators and operands that specifies a computation.*

---

digraphs In all aspects of the language, the six tokens[67)] 916

```
<:  :>  <%  %>  %:  %:%:
```

behave, respectively, the same as the six tokens

```
[   ]   {   }   #   ##
```

except for their spelling.[68)]

C90

These alternative spellings for some tokens were introduced in Amendment 1 to the C90 Standard. As such there is no change in behavior between C90 and C99.

6.4.7 Header names

Semantics

---

characters between < and >delimiters If the characters ', \, ", //, or /* occur in the sequence between the < and > delimiters, the behavior is undefined. 920

C90

The character sequence // was not specified as causing undefined behavior in C90 (which did not treat this sequence as the start of a comment).

---

921 Similarly, if the characters ', \, //, or /* occur in the sequence between the " delimiters, the behavior is undefined.69)

C90

The character sequence // was not specified as causing undefined behavior in C90 (which did not treat this sequence as the start of a comment).

---

924 ~~A header name preprocessing token is~~ Header name preprocessing tokens are recognized only within a **#include** preprocessing ~~directive.~~ directives or in implementation-defined locations within **#pragma** directives[DR324]. header name recognized within #include

C90

This statement summarizes the response to DR #017q39 against the C90 Standard.

C++

The C++ Standard contains the same wording as the C90 Standard.

6.4.8 Preprocessing numbers

---

927 pp-number syntax

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

C90

The C90 Standard used the syntax nonterminal *nondigit* rather than *identifier-nondigit*.

Rationale

C99 replaces *nondigit* with *identifier-nondigit* in the grammar to allow the token pasting operator, ##, to work as expected. Given the code

```
#define  mkident(s)  s ## 1m
/* ... */
int mkident(int) = 0;
```

if an identifier is passed to the mkident macro, then 1m is parsed as a single pp-number, a valid single identifier is produced by the ## operator, and nothing harmful happens. But consider a similar construction that might appear using Greek script:

```
#define  μμμμk(p)  p ## 1μ
/* ... */
int μk(int) = 0;
```

For this code to work, 1μ must be parsed as only one pp-token. Restricting pp-numbers to only the basic letters would break this.

Support for additional digits via UCNs is new in C99. Also support for `p` and `P` in a *pp-number* is new in C99.

C++

Support for `p` and `P` in a *pp-number* is new in C99 and is not specified in the C++ Standard.

Description

A preprocessing number begins with a digit optionally preceded by a period (`.`) and may be followed by valid identifier characters and the character sequences **`e+`**, **`e-`**, **`E+`**, **`E-`**, **`p+`**, **`p-`**, **`P+`**, or **`P-`**. 928

C90

Support for the `P` form of exponent is new in C99.

C++

The C++ Standard does not make this observation and like C90 does not support the `P` form of the exponent.

Preprocessing number tokens lexically include all floating and integer constant tokens. 929

C++

This observation is not made in the C++ Standard.

Semantics

6.4.9 Comments

---

comment /*

Except within a character constant, a string literal, or a comment, the characters `/*` introduce a comment. 934

C++

The C++ Standard does not explicitly specify the exceptions implied by the phases of translation.

---

comment contents only examined to

The contents of such a comment are examined only to identify multibyte characters and to find the characters `*/` that terminate it.[70] 935

C++

The C++ Standard gives no explicit meaning to any sequences of characters within a comment. It does call out the fact that comments do not nest and that the character sequence `//` is treated like any other character sequence within such a comment.

2.7p1 *The characters `/*` start a comment, which terminates with the characters `*/`.*

---

comment //

Except within a character constant, a string literal, or a comment, the characters `//` introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. 936

C90

Support for this style of comment is new in C99.

There are a few cases where a program's behavior will be altered by support for this style of commenting:

```
x = a //* */ b
      + c;

```

```
#define f(x) #x

f(a//) + g(
);
```

Occurrences of these constructs are likely to be rare.

C++

The C++ Standard does not explicitly specify the exceptions implied by the phases of translation.

---

937 The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

C++

The C++ Standard includes some restrictions on the characters that can occur after the characters //, which are not in

---

937 The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.
C90.

> *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.* 2.7p1

A C source file using the // style of comments may use form-feed or vertical-tab characters within that comment. Such a source file may not be acceptable to a C++ implementation. Occurrences of these characters within a comment are likely to be unusual.

6.5 Expressions

---

940 An *expression* is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof. expressions

C++

The C++ Standard (5p1) does not explicitly specify the possibility that an expression can designate an object or a function.

---

941 Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. DR287) object modified once between sequence points

C++

> *Between the previous and next sequence point a scalar object shall have its stored value modified at most once by the evaluation of an expression.* 5p4

The C++ Standard avoids any ambiguity in the interpretation of *object* by specifying scalar type.

---

945 Some operators (the unary operator ~, and the binary operators <<, >>, &, ^, and |, collectively described as *bitwise operators*) are required to have operands that have integer type. bitwise operators

C++

The C++ Standard does not define the term bitwise operators, although it does use the term bitwise in the description of the &, ^ and | operators.

bitwise operations signed types

These operators yield values that depend on the internal representations of integers, and have implementation-defined and undefined aspects for signed types. 946

C++

These operators exhibit the same range of behaviors in C++. This is called out within the individual descriptions of each operator in the C++ Standard.

---

exception condition

If an *exceptional condition* occurs during the evaluation of an expression (that is, if the result is not mathematically defined or not in the range of representable values for its type), the behavior is undefined. 947

C90

The term *exception* was defined in the C90 Standard, not *exceptional condition*.

C++

5p5 *If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined, unless such an expression is a constant expression (5.19), in which case the program is ill-formed.*

The C++ language contains explicit exception-handling constructs (Clause 15, **`try`/`throw`** blocks). However, these are not related to the mechanisms being described in the C Standard. The term exceptional condition is not defined in the C sense.

---

effective type

The *effective type* of an object for an access to its stored value is the declared type of the object, if any.[73)] 948

C90

The term *effective type* is new in C99.

C++

The term *effective type* is not defined in C++. A type needs to be specified when the C++ new operator is used. However, the C++ Standard includes the C library, so it is possible to allocate storage via a call to the `malloc` library function, which does not associate a type with the allocated storage.

Within each major subclause, the operators have the same precedence. 954

C++

This observation is true in the C++ Standard, but is not pointed out within that document.

---

footnote 73

73) Allocated objects have no declared type. 956

C90

The C90 Standard did not point this fact out.

C++

The C++ operator **`new`** allocates storage for objects. Its usage also specifies the type of the allocated object. The C library is also included in the C++ Standard, providing access to the `malloc` and `calloc` library functions (which do not contain a mechanism for specifying the type of the object created).

---

object value accessed if type

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:[74)] 960

C90

In the C90 Standard the term used in the following types was *derived type*. The term *effective type* is new in the C99 Standard and is used throughout the same list.

961 — a type compatible with the effective type of the object, object stored value accessed only by

C++

— the dynamic type of the object, 3.10p15

the type of the most derived object (1.8) to which the lvalue denoted by an lvalue expression refers. [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.] The dynamic type of an rvalue expression is its static type. 1.3.3 dynamic type

The difference between an object's dynamic and static type only has meaning in C++.

Use of effective type means that C gives types to some objects that have no type in C++. C++ requires the types to be the same, while C only requires that the types be compatible. However, the only difference occurs when an enumerated type and its compatible integer type are intermixed.

631 compatible type if

966 — a character type.

C++

*— a **char** or **unsigned char** type.* 3.10p15

The C++ Standard does not explicitly specify support for the character type **signed char**. However, it does specify that the type **char** may have the same representation and range of values as **signed char** (or **unsigned char**).

It is common practice to access the subcomponents of an object using a **char** or **unsigned char** type. However, there is code that uses **signed char**, and it would be a brave vendor whose implementation did not assume that objects having type **signed char** were not a legitimate alias for accesses to any object.

516 char range, representation and behavior

967 A floating expression may be *contracted*, that is, evaluated as though it were an atomic operation, thereby omitting rounding errors implied by the source code and the expression evaluation method.[75] contracted

C90

This explicit permission is new in C99.

C++

The C++ Standard, like C90, is silent on this subject.

968 The **FP_CONTRACT** pragma in **<math.h>** provides a way to disallow contracted expressions.

C90

Support for the `FP_CONTRACT` pragma is new in C99.

C++

Support for the `FP_CONTRACT` pragma is new in C99 and not specified in the C++ Standard.

969 Otherwise, whether and how expressions are contracted is implementation-defined.[76] contracted how implementation-defined

C++

The C++ Standard does not give implementations any permission to contract expressions. This does not mean they cannot contract expressions, but it does mean that there is no special dispensation for potentially returning different results.

---

footnote 75

75) A contracted expression might also omit the raising of floating-point exceptions. 972

C++

The contraction of expressions is not explicitly discussed in the C++ Standard.

---

footnote 76

76) This license is specifically intended to allow implementations to exploit fast machine instructions that combine multiple C operators. 973

C90

Such instructions were available in processors that existed before the creation of the C90 Standard and there were implementations that made use of them. However, this license was not explicitly specified in the C90 Standard.

C++

The C++ Standard contains no such explicit license.

6.5.1 Primary expressions

primary-expression syntax

975

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

C++

The C++ Standard (5.1p1) includes additional syntax that supports functionality not available in C.

Semantics

identifier is primary expression if

An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator).[77] 976

C++

The C++ definition of identifier (5.1p7) includes support for functionality not available in C. The C++ Standard uses the term *identifier functions*, not the term *function designator*. It also defines such identifier functions as being lvalues (5.2.2p10) but only if their return type is a reference (a type not available in C).

6.5.2 Postfix operators

postfix-expression syntax

985

```
postfix-expression:
                primary-expression
                postfix-expression [ expression ]
                postfix-expression ( argument-expression-list_opt )
                postfix-expression .  identifier
                postfix-expression -> identifier
                postfix-expression ++
                postfix-expression --
```

```
( type-name ) { initializer-list }
( type-name ) { initializer-list , }
```

argument-expression-list:

```
assignment-expression
argument-expression-list , assignment-expression
```

C90

Support for the forms (compound literals):

```
( type-name ) { initializer-list }
( type-name ) { initializer-list , }
```

is new in C99.

C++

Support for the forms (compound literals):

```
( type-name ) { initializer-list }
( type-name ) { initializer-list , }
```

is new in C99 and is not specified in the C++ Standard.

---

986 77) Thus, an undeclared identifier is a violation of the syntax. footnote 77

C++

The C++ Standard does not explicitly point out this consequence.

6.5.2.1 Array subscripting

Constraints

Semantics

---

992 If **E** is an *n*-dimensional array ($n \geq 2$) with dimensions $i \times j \times \cdots \times k$ then **E** (used as other than an lvalue) is converted to a pointer to an (*n*-1)-dimensional array with dimensions $j \times \cdots \times k$ array n-dimensional reference

C++

Clause 8.3.4p7 uses the term *rank* to describe $i \times j \times \cdots \times k$, not *dimensions*.

---

993 If the unary * operator is applied to this pointer explicitly, or implicitly as a result of subscripting, the result is the pointed-to (*n - 1*)-dimensional array, which itself is converted into a pointer if used as other than an lvalue.

C++

8.3.4p7

If the ∗ operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to ($n - 1$)-dimensional array, which itself is immediately converted into a pointer.

While the C++ Standard does not require the result to be used "as other than an lvalue" for it to be converted to a pointer. This difference does not result in any differences for the constructs available in C.

6.5.2.2 Function calls

Constraints

function call

The expression that denotes the called function[78] shall have type pointer to function returning **void** or returning an object type other than an array type. 997

C++

5.2.2p3 *This type shall be a complete object type, a reference type or the type* ***void****.*

Source developed using a C++ translator may contain functions returning an array type.

function call arguments agree with parameters

If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. 998

C++

C++ requires that all function definitions include a prototype.

5.2.2p6 *A function can be declared to accept fewer arguments (by declaring default arguments (8.3.6)) or more arguments (by using the ellipsis,* `...` *8.3.5) than the number of parameters in the function definition (8.4). [Note: this implies that, except where the ellipsis (*`...`*) is used, a parameter is available for each argument.]*

A called function in C++, whose definition uses the syntax specified in standard C, has the same restrictions placed on it by the C++ Standard as those in C.

argument type may be assigned

Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding parameter. 999

C++

The C++ language permits multiple definitions of functions having the same name. The process of selecting which function to call requires that a list of viable functions be created. Being a viable function requires:

13.3.2p3 *. . . , there shall exist for each argument an implicit conversion sequence that converts that argument to the corresponding parameter . . .*

A C source file containing a call that did not meet this criteria would cause a C++ implementation to issue a diagnostic (probably complaining about there not being a visible function declaration that matched the type of the call).

Semantics

operator ()

A postfix expression followed by parentheses `()` containing a possibly empty, comma-separated list of expressions is a function call. 1000

C90

The C90 Standard included the requirement:

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

extern int identifier();

appeared.

A C99 implementation will not perform implicit function declarations.

1004 In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument.[79]) function call preparing for

C++

The C++ Standard treats parameters as declarations that are initialized with the argument values:

When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument. 5.2.2p4

The behavior is different for arguments having a class type:

A class object can be copied in two ways, by initialization (12.1, 8.5), including for function argument passing (5.2.2) and for function value return (6.6.3), and by assignment (5.17). 12.8p1

This difference has no effect if a program is written using only the constructs available in C (structure and union types are defined by C++ to be class types).

1005 If the expression that denotes the called function has type pointer to function returning an object type, the function call expression has the same type as that object type, and has the value determined as specified in 6.8.6.4.

C90

A rather embarrassing omission from the original C90 Standard was the specification of the type of a function call. This oversight was rectified by the response to DR #017q37.

C++

Because C++ allows multiple function definitions having the same name, the wording of the return type is based on the type of function chosen to be called, not on the type of the expression that calls it.

The type of the function call expression is the return type of the statically chosen function . . . 5.2.2p3

1006 Otherwise, the function call has type **`void`**.

C++

C++ also supports functions returning reference types. This functionality is not available in C.

1007 If an attempt is made to modify the result of a function call or to access it after the next sequence point, the behavior is undefined. function result attempt to modify

C90

This sentence did not appear in the C90 Standard and had to be added to the C99 Standard because of a change in the definition of the term lvalue. 721 lvalue

C++

The C++ definition of lvalue is the same as that given in the C90 Standard, however, it also includes the wording:

5.2.2p10 *A function call is an lvalue if and only if the result type is a reference.*

In C++ it is possible to modify an object through a reference type returned as the result of a function call. Reference types are not available in C.

---

called function no prototype

If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **`float`** are promoted to **`double`**. 1008

C++

In C++ all functions must be defined with a type that includes a prototype.

A C source file that contains calls to functions that are declared without prototypes will be ill-formed in C++.

---

default argument promotions

These are called the *default argument promotions*. 1009

C++

The C++ Standard always requires a function prototype to be in scope at the point of call. However, it also needs to define default argument promotions (Clause 5.2.2p7) for use with arguments corresponding to the ellipsis notation.

---

argument in call incompatible with function definition

If the function is defined with a type that does not include a prototype, and the types of the arguments after promotion are not compatible with those of the parameters after promotion, the behavior is undefined, except for the following cases: 1012

C90

The C90 Standard did not include any exceptions.

C++

All functions must be defined with a type that includes a prototype.

A C source file that contains calls to functions that are declared without prototypes will be ill-formed in C++.

---

footnote 78

78) Most often, this is the result of converting an identifier that is a function designator. 1013

C++

The C++ language provides a mechanism for operators to be overloaded by developer-defined functions, creating an additional mechanism through which functions may be called. Although this mechanism is commonly discussed in textbooks, your author suspects that in practice it does not account for many function calls in C++ source code.

---

footnote 79

79) A function may change the values of its parameters, but these changes cannot affect the values of the arguments. 1014

C++

The C++ reference type provides a call by address mechanism. A change to a parameter declared to have such a type will immediately modify the value of its corresponding argument.

This C behavior also holds in C++ for all of the types specified by the C Standard.

On the other hand, it is possible to pass a pointer to an object, and the function may change the value of the object pointed to. 1015

C++

This possibility is not explicitly discussed in the C++ Standard, which supports an easier to use mechanism for modifying arguments, reference types.

1019 If the expression that denotes the called function has a type that does include a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters, taking the type of each parameter to be the unqualified version of its declared type. function call prototype visible

C90

The wording that specifies the use of the unqualified version of the parameter type was added by the response to DR #101.

C++

The C++ Standard specifies argument-passing in terms of initialization. For the types available in C, the effects are the same.

> *When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument.* 5.2.2p4

1020 The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter.

C++

There is no concept of starting and stopping, as such, argument conversion in C++.

1022 No other conversions are performed implicitly;

C++

In C++ it is possible for definitions written by a developer to cause implicit conversions. Such conversions can be applied to function arguments. C source code, given as input to a C++ translator, cannot contain any such constructs.

1023 in particular, the number and types of arguments are not compared with those of the parameters in a function definition that does not include a function prototype declarator.

C++

All function definitions must include a function prototype in C++.

1024 If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined. function call not compatible with definition

C++

In C++ is it possible for there to be multiple definitions of functions having the same name and different types. The process of selecting which function to call requires that a list of viable functions be created (it is possible that this selection process will not return any matching function).

It is possible that source developed using a C++ translator may contain pointer-to function conversions that happen to be permitted in C++, but have undefined behavior in C.

1025 The order of evaluation of the function designator, the actual arguments, and subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call. function call sequence point

C++

5.2.2p8 *All side effects of argument expression evaluations take effect before the function is entered.*

While the requirements in the C++ Standard might appear to be a subset of the requirements in the C Standard, they are effectively equivalent. The C Standard does not require that the evaluation of any other operands, which may occur in the expression containing the function call, have occurred prior to the call and the C++ Standard does not prohibit them from occurring.

6.5.2.3 Structure and union members

Constraints

Semantics

A postfix expression followed by the -> operator and an identifier designates a member of a structure or union object. 1034

C++

The C++ Standard specifies how the operator can be mapped to the dot (`.`) form and describes that operator only.

5.2.5p3 *If `E1` has the type "pointer to class X," then the expression `E1->E2` is converted to the equivalent form `(*(E1)).E2`; the remainder of 5.2.5 will address only the first option (dot)[59].*

---

union special guarantee

One special guarantee is made in order to simplify the use of unions: if a union contains several structures that share a common initial sequence (see below), and if the union object currently contains one of these structures, it is permitted to inspect the common initial part of any of them anywhere that a declaration of the complete type of the union is visible. 1037

C90

The wording:

anywhere that a declaration of the complete type of the union is visible.

EXAMPLE member selection 1044
value stored in union 586

was added in C99 to handle a situation that was raised too late in the process to be published in a Technical Report. Another wording change relating to accessing members of union objects is discussed elsewhere.

C++

Like C90, the C++ Standard does not include the words " . . . anywhere that a declaration of the complete type of the union is visible."

---

common initial sequence

Two structures share a *common initial sequence* if corresponding members have compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members. 1038

C++

3.9p11 *If two types T1 and T2 are the same type, then T1 and T2 are layout-compatible types.*

9.2p16

Two POD-structs share a common initial sequence if corresponding members have layout-compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

POD is an acronym for *Plain Old Data* type.
C++ requires that types be the same, while C requires type compatibility. If one member has an enumerated type and its corresponding member has the compatible integer type, C can treat these members as being part of a common initial sequence. C++ will not treat these members as being part of a common initial sequence.

1041 80) If **&E** is a valid pointer expression (where **&** is the "address-of" operator, which generates a pointer to its operand), the expression **(&E)->MOS** is the same as **E.MOS**. footnote 80

C++

The C++ Standard does not make this observation.

1044 EXAMPLE 3 The following is a valid fragment: EXAMPLE member selection

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

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

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

C90

In C90 the second fragment was considered to contain implementation-defined behavior.

C++

The behavior of this example is as well defined in C++ as it is in C90.

6.5.2.4 Postfix increment and decrement operators

Constraints

---

postfix operator constraint

The operand of the postfix increment or decrement operator shall have qualified or unqualified real or pointer type and shall be a modifiable lvalue. 1046

C90

The C90 Standard required the operand to have a scalar type.

C++

D.1p1 *The use of an operand of type **bool** with the postfix **++** operator is deprecated.*

Semantics

After the result is obtained, the value of the operand is incremented. 1048

C++

5.2.6p1 *After the result is noted, the value of the object is modified by adding 1 to it, unless the object is of type **bool**, in which case it is set to **true**. [Note: this use is deprecated, see annex D.]*

_Bool 476 large enough to store 0 and 1

The special case for operands of type **bool** also occurs in C, but a chain of reasoning is required to deduce it.

---

postfix operators see also

See the discussions of additive operators and compound assignment for information on constraints, types, and conversions and the effects of operations on pointers. 1050

C++

The C++ Standard provides a reference, but no explicit wording that the conditions described in the cited clauses also apply to this operator.

5.2.6p1 *See also 5.7 and 5.17.*

The side effect of updating the stored value of the operand shall occur between the previous and the next sequence point. 1051

C++

sequence points

The C++ Standard does not explicitly specify this special case of a more general requirement.

---

postfix -- analogous to ++

The postfix -- operator is analogous to the postfix ++ operator, except that the value of the operand is decremented (that is, the value 1 of the appropriate type is subtracted from it). 1052

C++

5.2.6p2

> *. . . except that the operand shall not be of type **bool**.*

A C source file containing an instance of the postfix `--` operator applied to an operand having type `_Bool` is likely to result in a C++ translator issuing a diagnostic.

6.5.2.5 Compound literals

1053 **Forward references:** additive operators (6.5.6), compound assignment (6.5.16.2).

C90

Support for compound literals is new in C99.

C++

Compound literals are new in C99 and are not available in C++.

Constraints

Semantics

6.5.3 Unary operators

6.5.3.1 Prefix increment and decrement operators

Constraints

1081 The operand of the prefix increment or decrement operator shall have qualified or unqualified real or pointer type and shall be a modifiable lvalue. postfix operator operand

C++

The use of an operand of type **bool** with the prefix `++` operator is deprecated (5.3.2p1); there is no corresponding entry in Annex D, but the proposed response to C++ DR #145 inserted one. In the case of the decrement operator:

> *The operand shall not be of type **bool**.* 5.3.2p1

A C source file containing an instance of the prefix `--` operator applied to an operand having type `_Bool` is likely to result in a C++ translator issuing a diagnostic.

Semantics

1084 The expression `++E` is equivalent to `(E+=1)`.

C++

C++ lists an exception (5.3.2p1) for the case when `E` has type **bool**. This is needed because C++ does not define its boolean type in the same way as C. The behavior of this operator on operands is defined as a special case in C++. The final result is the same as in C. 476 _Bool large enough to store 0 and 1

1085 See the discussions of additive operators and compound assignment for information on constraints, types, side effects, and conversions and the effects of operations on pointers. prefix operators see also

C++

> *[Note: see the discussions of addition (5.7) and assignment operators (5.17) for information on conversions.]* 5.3.2p1

There is no mention that the conditions described in these clauses also apply to this operator.

The prefix -- operator is analogous to the prefix ++ operator, except that the value of the operand is decremented. 1086

C++

The prefix **--** operator is not analogous to the prefix **++** operator in that its operand may not have type **bool**.

6.5.3.2 Address and indirection operators

Constraints

unary & operand constraints

The operand of the unary & operator shall be either a function designator, the result of a [] or unary * operator, or an lvalue that designates an object that is not a bit-field and is not declared with the **register** storage-class specifier. 1088

C90

The words:

> *. . . , the result of a* **[ ]** *or unary* * *operator,*

are new in C99 and were added to cover the following case:

```
int a@lsquare[]10@rsquare[];

for (int *p = &a@lsquare[]0@rsquare[]; p < &a@lsquare[]10@rsquare[]; p++)
    /* ... */
```

where C90 requires the operand to refer to an object. The expression a+10 exists, but does not refer to an object. In C90 the expression &a[10] is undefined behavior, while C99 defines the behavior.

C++

Like C90 the C++ Standard does not say anything explicit about the result of a **[]** or unary * operator. The C++ Standard does not explicitly exclude objects declared with the **register** storage-class specifier appearing as operands of the unary **&** operator. In fact, there is wording suggesting that such a usage is permitted:

7.1.1p3

> *A* ***register*** *specifier has the same semantics as an* ***auto*** *specifier together with a hint to the implementation that the object so declared will be heavily used. [Note: the hint can be ignored and in most implementations it will be ignored if the address of the object is taken. —end note]*

Source developed using a C++ translator may contain occurrences of the unary **&** operator applied to an operand declared with the **register** storage-class specifier, which will cause a constraint violation if processed by a C translator.

```
void f(void)
{
register int a@lsquare[]10@rsquare[]; /* undefined behavior */
                    // well-formed

&a@lsquare[]1@rsquare[]  /* constraint violation */
       // well-formed
     ;
}
```

unary * operand has pointer type

The operand of the unary * operator shall have pointer type. 1089

C++

> *The unary * operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type . . .* 5.3.1p1

C++ does not permit the unary * operator to be applied to an operand having a pointer to **void** type.

```
void *g_ptr;

void f(void)
{
&*g_ptr; /* DR #012 */
         // DR #232
}
```

Semantics

1090 The unary & operator yields the address of its operand. unary & operator

C90

This sentence is new in C99 and summarizes what the unary & operator does.

C++

Like C90, the C++ Standard specifies a pointer to its operand (5.3.1p1). But later on (5.3.1p2) goes on to say: "In particular, the address of an object of type "*cv* T" is "pointer to *cv* T," with the same cv-qualifiers."

1092 If the operand is the result of a unary * operator, neither that operator nor the & operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. &*

C90

The responses to DR #012, DR #076, and DR #106 specified that the above constructs were constraint violations. However, no C90 implementations known to your author diagnosed occurrences of these constructs.

C++

This behavior is not specified in C++. Given that either operator could be overloaded by the developer to have a different meaning, such a specification would be out of place.

At the time of this writing a response to C++ DR #232 is being drafted (a note from the Oct 2003 WG21 meeting says: "We agreed that the approach in the standard seems okay: `p = 0; *p;` is not inherently an error. An lvalue-to-rvalue conversion would give it undefined behavior.").

```
void DR_232(void)
{
int *loc = 0;

if (&*loc == 0) /* no dereference of a null pointer, defined behavior */
                // probably not a dereference of a null pointer.
   ;

&*loc = 0; /* not an lvalue in C */
           // how should an implementation interpret the phrase must not (5.3.1p1)?
}
```

Similarly, if the operand is the result of a `[]` operator, neither the `&` operator nor the unary `*` that is implied by the `[]` is evaluated and the result is as if the `&` operator were removed and the `[]` operator were changed to a `+` operator. 1093

C90

This requirement was not explicitly specified in the C90 Standard. It was the subject of a DR #076 that was closed by adding this wording to the C99 Standard.

C++

This behavior is not specified in C++. Given that either operator could be overloaded by the developer to have a different meaning, such a specification would be out of place. The response to C++ DR #232 may specify the behavior for this case.

unary * indirection

The unary `*` operator denotes indirection. 1095

C++

5.3.1p1 *The unary `*` operator performs indirection.*

If the operand points to a function, the result is a function designator; 1096

C++

The C++ Standard also specifies (5.3.1p1) that this result is an lvalue. This difference is only significant for reference types, which are not supported by C.

If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined.[84)] 1099

C++

The C++ Standard does not explicitly state the behavior for this situation.

6.5.3.3 Unary arithmetic operators

Constraints

The operand of the unary `+` or `-` operator shall have arithmetic type; 1101

C++

The C++ Standard permits the operand of the unary `+` operator to have pointer type (5.3.1p6).

! operand type

of the `!` operator, scalar type. 1103

C++

The C++ Standard does not specify any requirements on the type of the operand of the `!` operator.

5.3.1p8 *The operand of the logical negation operator `!` is implicitly converted to **`bool`** (clause 4);*

But the behavior is only defined if operands of scalar type are converted to **`bool`**:

4.12p1

> *An rvalue of arithmetic, enumeration, pointer, or pointer to member type can be converted to an rvalue of type* ***bool****.*

Semantics

1110 If the promoted type is an unsigned type, the expression `~E` is equivalent to the maximum value representable in that type minus `E`.

C++

The C++ Standard does not point out this equivalence.

1111 The result of the logical negation operator `!` is 0 if the value of its operand compares unequal to 0, 1 if the value of its operand compares equal to 0. logical negation result is

C++

> *its value is* ***true*** *if the converted operand is* ***false*** *and* ***false*** *otherwise.* 5.3.1p8

This difference is only visible to the developer in one case. In all other situations the behavior is the same false and true will be converted to 0 and 1 as-needed. 1112 logical negation result type

1112 The result has type `int`. logical negation result type

C++

> *The type of the result is* ***bool****.* 5.3.1p8

The difference in result type will result in a difference of behavior if the result is the immediate operand of the `sizeof` operator. Such usage is rare.

1113 The expression `!E` is equivalent to `(0==E)`. ! equivalent to

C++

There is no explicit statement of equivalence given in the C++ Standard.

1114 84) Thus, `&*E` is equivalent to `E` (even if `E` is a null pointer), and `&(E1[E2])` to `((E1)+(E2))`. footnote 84

C90

This equivalence was not supported in C90, as discussed in the response to DR #012, #076, and #106.

C++

At the moment the C++ Standard specifies no such equivalence, explicitly or implicitly. However, this situation may be changed by the response to DR #232.

1116 If `*P` is an lvalue and `T` is the name of an object pointer type, `*(T)P` is an lvalue that has a type compatible with that to which `T` points.

C++

The C++ Standard makes no such observation.

Among the invalid values for dereferencing a pointer by the unary `*` operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime. 1117

C90

The wording in the C90 Standard only dealt with the address of objects having automatic storage duration.

C++

The C++ Standard does not call out a list of possible invalid values that might be dereferenced.

6.5.3.4 The `sizeof` operator

Constraints

sizeof constraints

The **sizeof** operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field member. 1118

C++

The C++ Standard contains a requirement that does not exist in C.

5.3.3p5 *Types shall not be defined in a **sizeof** expression.*

A C source file that defines a type within a **sizeof** expression is likely to cause a C++ translator to issue a diagnostic. Defining a type within a **sizeof** expression is rarely seen in C source.

```
int glob = sizeof(enum {E1, E2}); /* does not affect the conformance status of the program */
                                  // ill-formed
```

Semantics

The size is determined from the type of the operand. 1120

C++

5.3.3p1 *The **sizeof** operator yields the number of bytes in the object representation of its operand.*

The result is an integer. 1121

C90

In C90 the result was always an integer constant. The C99 contexts in which the result is not an integer constant all involve constructs that are new in C99.

C++

Like C90, the C++ Standard specifies that the result is a constant. The cases where the result is not a constant require the use of types that are not supported by C++.

sizeof operand evaluated

If the type of the operand is a variable length array type, the operand is evaluated; 1122

C90

Support for variable length array types is new in C99.

C++

Variable length array types are new in C99 and are not available in C++.

1127 The value of the result is implementation-defined, and its type (an unsigned integer type) is **size_t**, defined in **<stddef.h>** (and other headers). sizeof result type

C++

> *. . . ; the result of* **sizeof** *applied to any other fundamental type (3.9.1) is implementation-defined.* 5.3.3p1

The C++ Standard does not explicitly specify any behavior when the operand of **sizeof** has a derived type. A C++ implementation need not document how the result of the **sizeof** operator applied to a derived type is calculated.

1130 EXAMPLE 3 In this example, the size of a variable length array is computed and returned from a function:

```
#include <stddef.h>

size_t fsize3(int n)
{
        char b[n+3];       // variable length array
        return sizeof b;   // execution time sizeof
}

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

C90

This example, and support for variable length arrays, is new in C99.

1131 85) When applied to a parameter declared to have array or function type, the **sizeof** operator yields the size of the adjusted (pointer) type (see 6.9.1). footnote 85

C++

This observation is not made in the C++ Standard.

6.5.4 Cast operators

1133 cast-expression syntax

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

C++

The C++ Standard uses the terminal name *type-id*, not *type-name*.

Constraints

1134 Unless the type name specifies a void type, the type name shall specify qualified or unqualified scalar type and the operand shall have scalar type. cast scalar or void type

C++

There is no such restriction in C++ (which permits the type name to be a class type). However, the C++ Standard contains a requirement that does not exist in C.

5.4p3 *Types shall not be defined in casts.*

A C source file that defines a type within a cast is likely to cause a C++ translator to issue a diagnostic (this usage is rare).

```
extern int glob;

void f(void)
{
switch ((enum {E1, E2, E3})glob) /* does not affect the conformance status of the program */
                                 // ill-formed
   {
   case E1: glob+=3;
            break;
   /* ... */
   }
}
```

---

pointer conversion constraints

Conversions that involve pointers, other than where permitted by the constraints of 6.5.16.1, shall be specified by means of an explicit cast. 1135

C90

This wording appeared in the Semantics clause in the C90 Standard; it was moved to constraints in C99. This is not a difference if it is redundant.

C++

The C++ Standard words its specification in terms of assignment:

4p3 *An expression* `e` *can be implicitly converted to a type* `T` *if and only if the declaration "*`T t=e;`*" is well-formed, for some invented temporary variable* `t` *(8.5).*

Semantics

---

cast

This construction is called a *cast*.[86] 1137

C++

The C++ Standard uses the phrase *cast notation*. There are other ways of expressing a type conversion in C++ (functional notation, or a type conversion operator). The word *cast* could be said to apply in common usage to any of these forms (when technically it refers to none of them).

A cast that specifies no conversion has no effect on the type or value of an expression.[87] 1138

C++

The C++ Standard explicitly permits an expression to be cast to its own type (5.2.11p1), but does not list any exceptions for such an operation.

---

footnote 86

86) A cast does not yield an lvalue. 1140

C++

The result is an lvalue if T is a reference type, otherwise the result is an rvalue. 5.4p1

Reference types are not available in C, so this specification is not a difference in behavior for a conforming C program.

1141 Thus, a cast to a qualified type has the same effect as a cast to the unqualified version of the type.

C++

Casts that involve qualified types can be a lot more complex in C++ (5.2.11). There is a specific C++ cast notation for dealing with this form of type conversion, `const_cast<T>` (where `T` is some type).

[Note: some conversions which involve only changes in cv-qualification cannot be done using ***`const_cast`****. For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior.* 5.2.11p12

The other forms of conversions involve types not available in C.

---

footnote 87

1142 ~~87)~~ If the value of the expression is represented with greater precision or range than required by the type named by the cast (6.3.1.8), then the cast specifies a conversion even if the type of the expression is the same as the named type.

C++

The C++ Standard is silent on this subject.

6.5.5 Multiplicative operators

---

multiplicative-expression syntax

1143

```
multiplicative-expression:
            cast-expression
            multiplicative-expression * cast-expression
            multiplicative-expression / cast-expression
            multiplicative-expression % cast-expression
```

C++

In C++ there are two operators (pointer-to-member operators, `.*` and `->*`) that form an additional precedence level between *`cast-expression`* and *`multiplicative-expression`*. The nonterminal name for such expressions is *`pm-expression`*, which appears in the syntax for *`multiplicative-expression`*.

Constraints

Semantics

1151 When integers are divided, the result of the `/` operator is the algebraic quotient with any fractional part discarded.[88)]

C90

When integers are divided and the division is inexact, if both operands are positive the result of the **`/`** *operator is the largest integer less than the algebraic quotient and the result of the* **`%`** *operator is positive. If either operand is negative, whether the result of the* **`/`** *operator is the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient is implementation-defined, as is the sign of the result of the* **`%`** *operator.*

If either operand is negative, the behavior may differ between C90 and C99, depending on the implementation-defined behavior of the C90 implementation.

```
#include <stdio.h>

int main(void)
{
int x = -1,
    y = +3;

if ((x%y > 0) ||
   ((x+y)%y == x%y))
   printf("This is a C90 translator behaving differently than C99\n");
}
```

Quoting from the C9X Revision Proposal, WG14/N613, that proposed this change:

WG14/N613 *The origin of this practice seems to have been a desire to map C's division directly to the "natural" behavior of the target instruction set, whatever it may be, without requiring extra code overhead that might be necessary to check for special cases and enforce a particular behavior. However, the argument that Fortran programmers are unpleasantly surprised by this aspect of C and that there would be negligible impact on code efficiency was accepted by WG14, who agreed to require Fortran-like behavior in C99.*

C++

5.6p4 *If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined[74).]*

Footnote 74 describes what it calls *the preferred algorithm* and points out that this algorithm follows the rules by the Fortran Standard and that C99 is also moving in that direction (work on C99 had not been completed by the time the C++ Standard was published).

The C++ Standard does not list any options for the implementation-defined behavior. The most likely behaviors are those described by the C90 Standard (see C90/C99 difference above).

6.5.6 Additive operators

Constraints

---

addition operand types

For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to an object type and the other shall have integer type. 1154

C++

The C++ Standard specifies that the null pointer constant has an integer type that evaluates to zero (4.10p1). In C the `NULL` macro might expand to an expression having a pointer type. The expression `NULL+0` is always a null pointer constant in C++, but it may violate this constraint in C. This difference will only affect C source developed using a C++ translator and subsequently translated with a C translator that defines the `NULL` macro to have a pointer type (occurrences of such an expression are also likely to be very rare).

---

footnote 88

88) This is often called "truncation toward zero". 1157

C90

This term was not defined in the C90 Standard because it was not necessarily the behavior, for this operator, performed by an implementation.

C++

Footnote 74 uses the term *rounded toward zero*.

1159 — both operands are pointers to qualified or unqualified versions of compatible object types; or subtraction pointer operands

C++

> — *both operands are pointers to cv-qualified or cv-unqualified versions of the same completely defined object type; or* 5.7p2

Requiring the same type means that a C++ translator is likely to issue a diagnostic if an attempt is made to subtract a pointer to an enumerated type from its compatible integer type (or vice versa). The behavior is undefined in C if the pointers don't point at the same object.

```
#include <stddef.h>

enum e_tag {E1, E2, E3}; /* Assume compatible type is int. */
union {
      enum e_tag m_1@lsquare[]5@rsquare[];
      int m_2@lsquare[]10@rsquare[];
      } glob;

extern enum e_tag *p_e;
extern int *p_i;

void f(void)
{
ptrdiff_t loc = p_i-p_e; /* does not affect the conformance status of the program */
                         // ill-formed
}
```

The expression NULL-0 is covered by the discussion on operand types for addition. 1154 addition operand types

Semantics

1172 If the result points one past the last element of the array object, it shall not be used as the operand of a unary * operator that is evaluated. one past the end accessing

C++

This requirement is not explicitly specified in the C++ Standard.

1173 When two pointers are subtracted, both shall point to elements of the same array object, or one past the last element of the array object; pointer subtraction point at same object

C90

The C90 Standard did not include the wording "or one past the last element of the array object;". However, all implementations known to your author handled this case according to the C99 specification. Therefore, it is not listed as a difference.

1176 If the result is not representable in an object of that type, the behavior is undefined.

C90

As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined.

EXAMPLE Pointer arithmetic is well defined with pointers to variable length array types. 1179

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

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

C90

This example, and support for variable length arrays, is new in C99.

6.5.7 Bitwise shift operators

Constraints

Semantics

If **E1** has a signed type and nonnegative value, and **E1** $\times 2^{E2}$ is representable in the result type, then that is the resulting value; 1192

C90

This specification of behavior is new in C99; however, it is the behavior that all known C90 implementations exhibit.

C++

Like the C90 Standard, the C++ Standard says nothing about this case.

---

left-shift undefined

otherwise, the behavior is undefined. 1193

C90

This undefined behavior was not explicitly specified in the C90 Standard.

C++

Like the C90 Standard, the C++ Standard says nothing about this case.

6.5.8 Relational operators

Constraints

---

relational operators real operands

— both operands have real type; 1199

C90

both operands have arithmetic type;

The change in terminology in C99 was necessitated by the introduction of complex types.

1200 — both operands are pointers to qualified or unqualified versions of compatible object types; or relational operators pointer operands

C++

> *Pointers to objects or functions of the same type (after pointer conversions) can be compared, with a result defined as follows:* 5.9p2

The pointer conversions (4.4) handles differences in type qualification. But the underlying basic types have to be the same in C++. C only requires that the types be compatible. When one of the pointed-to types is an enumerated type and the other pointed-to type is the compatible integer type, C permits such operands to occur in the same relational-expression; C++ does not (see pointer subtraction for an example). 1159 subtraction pointer operands

1201 — both operands are pointers to qualified or unqualified versions of compatible incomplete types. relational operators pointer to incomplete type

C++

C++ classifies incomplete object types that can be completed as object types, so the discussion in the previous C sentence is also applicable here. 475 object types

Semantics

1203 For the purposes of these operators, a pointer to an object that is not an element of an array behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type. relational operators pointer to object

C++

This wording appears in 5.7p4, Additive operators, but does not appear in 5.9, Relational operators. This would seem to be an oversight on the part of the C++ committee, as existing implementations act as if the requirement was present in the C++ Standard.

1204 When two pointers are compared, the result depends on the relative locations in the address space of the objects pointed to.

C++

The C++ Standard does not make this observation.

1205 If two pointers to object or incomplete types both point to the same object, or both point one past the last element of the same array object, they compare equal.

C++

This requirement can be deduced from:

> — *If two pointers `p` and `q` of the same type point to the same object or function, or both point one past the end of the same array, or are both null, then `p<=q` and `p>=q` both yield **`true`** and `p<q` and `p>q` both yield **`false`**.* 5.9p2

1206 If the objects pointed to are members of the same aggregate object, pointers to structure members declared later compare greater than pointers to members declared earlier in the structure, and pointers to array elements with larger subscript values compare greater than pointers to elements of the same array with lower subscript values. structure members later compare later array elements later compare later

C++

This requirement also applies in C++ (5.9p2). If the declaration of two pointed-to members are separated by an access-specifier label (a construct not available in C), the result of the comparison is unspecified.

If the expression **P** points to an element of an array object and the expression **Q** points to the last element of the same array object, the pointer expression **Q+1** compares greater than **P**. 1208

C90

The C90 Standard contains the additional words, after those above:

> *even though Q+1 does not point to an element of the array object.*

relational pointer comparison undefined if not same object

In all other cases, the behavior is undefined. 1209

C90

If the objects pointed to are not members of the same aggregate or union object, the result is undefined with the following exception.

C++

> 5.9p2 — *Other pointer comparisons are unspecified.*

Source developed using a C++ translator may contain pointer comparisons that would cause undefined behavior if processed by a C translator.

relational operators result value

Each of the operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) shall yield 1 if the specified relation is true and 0 if it is false.[90] 1210

C++

> 5.9p1 *The operators* **<** *(less than),* **>** *(greater than),* **<=** *(less than or equal to), and* **>=** *(greater than or equal to) all yield* ***false*** *or* ***true****.*

relational operators result type 1211

This difference is only visible to the developer in one case, the result type. In all other situations the behavior is the same; false and true will be converted to 0 and 1 as-needed.

relational operators result type

The result has type **int**. 1211

C++

> 5.9p1 *The type of the result is* ***bool****.*

The difference in result type will result in a difference of behavior if the result is the immediate operand of the **sizeof** operator. Such usage is rare.

6.5.9 Equality operators

Constraints

equality operators constraints

One of the following shall hold: 1213

C++

5.10p1

> *The **==** (equal to) and the **!=** (not equal to) operators have the same semantic restrictions, conversions, and result type as the relational operators . . .*

See relational operators. relational operators constraints

1215 — both operands are pointers to qualified or unqualified versions of compatible types; equality operators pointer to compatible types

C++

The discussion on the relational operators is applicable here. 1200 relational operators pointer operands

---

1216 — one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**; or equality operators pointer to incomplete type

C++

This special case is not called out in the C++ Standard.

```
#include <stdlib.h>

struct node {
            int mem;
            };
void *glob;

void f(void)
{
/* The following is conforming */
// The following is ill-formed
struct node *p = malloc(sizeof(struct node));

/*
 * There are no C/C++ differences when the object being assigned
 * has a pointer to void type, 4.10p2.
 */
glob = p;
}
```

See relational operators for additional issues. relational operators constraints

Semantics

1219 Each of the operators yields 1 if the specified relation is true and 0 if it is false. equality operators true or false

C++

> *The **==** (equal to) and the **!=** (not equal to) operators have the same . . . truth-value result as the relational operators.* 5.10p1

This difference is only visible to the developer in one case. In all other situations the behavior is the same—false and true will be converted to 0 and 1 as needed. 1220 equality operators result type

---

1220 The result has type **int**. equality operators result type

C++

5.10p1

The == (equal to) and the != (not equal to) operators have the same . . . result type as the relational operators.

relational operators result type 1211

The difference is also the same as relational operators.

---

equality operators exactly one relation is true

For any pair of operands, exactly one of the relations is true. 1221

C90

This requirement was not explicitly specified in the C90 Standard. It was created, in part, by the response to DR #172.

C++

This requirement is not explicitly specified in the C++ Standard.

If both of the operands have arithmetic type, the usual arithmetic conversions are performed. 1222

C90

Where the operands have types and values suitable for the relational operators, the semantics detailed in 6.3.8 apply.

Values of complex types are equal if and only if both their real parts are equal and also their imaginary parts are equal. 1223

C90

Support for complex types is new in C99.

Any two values of arithmetic types from different type domains are equal if and only if the results of their conversions to the (complex) result type determined by the usual arithmetic conversions are equal. 1224

C90

Support for different type domains, and complex types, is new in C99.

C++

real type converted to complex 700

The concept of type domain is new in C99 and is not specified in the C++ Standard, which defines constructors to handle this case. The conversions performed by these constructions have the same effect as those performed in C.

---

footnote 91

91) The expression **a<b<c** is not interpreted as in ordinary mathematics. 1225

C++

The C++ Standard does not make this observation.

Otherwise, at least one operand is a pointer. 1229

C++

The C++ Standard does not break its discussion down into the nonpointer and pointer cases.

---

equality operators null pointer constant converted

If one operand is a pointer and the other is a null pointer constant, the null pointer constant is converted to the type of the pointer. 1230

C90

> *If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type.*

In the case of the expression `(void *)0 == 0` both operands are null pointer constants. The C90 wording permits the left operand to be converted to the type of the right operand (type **`int`**). The C99 wording does not support this interpretation. 748 null pointer constant

C++

The C++ Standard supports this combination of operands but does not explicitly specify any sequence of operations that take place prior to the comparison. equality operators null pointer constant

---

1231 If one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **`void`**, the former is converted to the type of the latter. equality operators pointer to void

C++

This conversion is part of the general pointer conversion (4.10) rules in C++. This conversion occurs when two operands have pointer type.

1233 Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.[92)] pointers compare equal

C90

> *If two pointers to object or incomplete types are both null pointers, they compare equal. If two pointers to object or incomplete types compare equal, they both are null pointers, or both point to the same object, or both point one past the last element of the same array object. If two pointers to function types are both null pointers or both point to the same function, they compare equal. If two pointers to function types compare equal, either both are null pointers, or both point to the same function.*

The admission that a pointer one past the end of an object and a pointer to the start of a different object compare equal, if the implementation places the latter immediately following the former in the address space, is new in C99 (but it does describe the behavior of most C90 implementations).

C++

> *Two pointers of the same type compare equal if and only if they are both null, both point to the same object or function, or both point one past the end of the same array.* 5.10p1

This specification does not include the cases:

- "(including a pointer to an object and a subobject at its beginning)", which might be deduced from wording given elsewhere, 761 object lowest addressed byte
- "or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space".

The C++ Standard does not prevent an implementation from returning a result of **`true`** for the second case, but it does not require it. However, the response to C++ DR #073 deals with the possibility of a pointer pointing one past the end of an object comparing equal, in some implementations, to the address of another object. Wording changes are proposed that acknowledge this possibility.

6.5.10 Bitwise AND operator

Constraints

& binary operand type

Each of the operands shall have integer type. 1235

C++

The wording of the specification in the C++ Standard is somewhat informal (the same wording is given for the bitwise exclusive-OR operator, 5.12p1, and the bitwise inclusive-OR operator, 5.13p1).

5.11p1 *The operator applies only to integral or enumeration operands.*

Semantics

& binary operands converted

The usual arithmetic conversions are performed on the operands. 1236

C++

The following conversion is presumably performed on the operands.

5.11p1 *The usual arithmetic conversions are performed;*

---

footnote 92

92) Two objects may be adjacent in memory because they are adjacent elements of a larger array or adjacent members of a structure with no padding between them, or because the implementation chose to place them so, even though they are unrelated. 1238

C90

The C90 Standard did not discuss these object layout possibilities.

C++

The C++ Standard does not make these observations.

If prior invalid pointer operations (such as accesses outside array bounds) produced undefined behavior, subsequent comparisons also produce undefined behavior. 1239

C90

The C90 Standard did not discuss this particular case of undefined behavior.

6.5.11 Bitwise exclusive OR operator

Constraints

Semantics

6.5.12 Bitwise inclusive OR operator

Constraints

Semantics

6.5.13 Logical AND operator

Constraints

1249 Each of the operands shall have scalar type. &&
operand type

C++

5.14p1

> *The operands are both implicitly converted to type* **`bool`** *(clause 4).*

Boolean conversions (4.12) covers conversions for all of the scalar types and is equivalent to the C behavior.

Semantics

1250 The `&&` operator shall yield 1 if both of its operands compare unequal to 0; &&
operand compare against 0

C++

5.14p1

> *The result is* **`true`** *if both operands are* **`true`** *and* **`false`** *otherwise.*

The difference in operand types is not applicable because C++ defines equality to return **`true`** or **`false`**. The difference in return value will not cause different behavior because **`false`** and **`true`** will be converted to 0 and `1` when required.

1252 The result has type **`int`**. &&
result type

C++

5.14p2

> *The result is a* **`bool`**.

The difference in result type will result in a difference of behavior if the result is the immediate operand of the **`sizeof`** operator. Such usage is rare.

1254 there is a sequence point after the evaluation of the first operand. &&
sequence point

C++

5.14p2

> *All side effects of the first expression except for destruction of temporaries (12.2) happen before the second expression is evaluated.*

The possible difference in behavior is the same as for the function-call operator. 1025 function call sequence point

6.5.14 Logical OR operator

Constraints

1257 Each of the operands shall have scalar type.

C++

5.15p1

*The operands are both implicitly converted to **bool** (clause 4).*

Boolean conversions (4.12) covers conversions for all of the scalar types and is equivalent to the C behavior.

Semantics

|| operand compared against 0

The || operator shall yield 1 if either of its operands compare unequal to 0; 1258

C++

5.15p1 *It returns **true** if either of its operands is **true**, and **false** otherwise.*

The difference in operand types is not applicable because C++ defines equality to return **true** or **false**. The difference in return value will not cause different behavior because **false** and **true** will be converted to 0 and 1 when required.

|| result type

The result has type **int**. 1260

C++

5.15p2 *The result is a **bool**.*

The difference in result type will result in a difference of behavior if the result is the immediate operand of the **sizeof** operator. Such usage is rare.

operator || sequence point

there is a sequence point after the evaluation of the first operand. 1262

C++

5.15p2 *All side effects of the first expression except for destruction of temporaries (12.2) happen before the second expression is evaluated.*

&& sequence point 1254 The differences are discussed elsewhere.

6.5.15 Conditional operator

conditional-expression syntax

1264

```
conditional-expression:
        logical-OR-expression
        logical-OR-expression ? expression : conditional-expression
```

C++

5.16 `conditional-expression: logical-or-expression logical-or-expression ? expression : assignment-expression`

assignment-expression syntax 1288 By supporting an *assignment-expression* as the third operand, C++ enables the use of a *throw-expression*; for instance:

```
z = can_I_deal_with_this() ? 42 : throw X;
```

Source developed using a C++ translator may contain uses of the conditional operator that are a constraint violation if processed by a C translator. For instance, the expression `x?a:b=c` will need to be rewritten as `x?a:(b=c)`.

Constraints

1265 The first operand shall have scalar type.

C++

*The first expression is implicitly converted to **`bool`** (clause 4).* 5.16p1

Boolean conversions (4.12) covers conversions for all of the scalar types and is equivalent to the C behavior.

1270 — both operands are pointers to qualified or unqualified versions of compatible types; conditional expression pointer to compatible types

C++

— The second and third operands have pointer type, or one has pointer type and the other is a null pointer constant; pointer conversions (4.10) and qualification conversions (4.4) are performed to bring them to their composite pointer type (5.9). 5.16p6

These conversions will not convert a pointer to an enumerated type to a pointer to integer type.
If one pointed-to type is an enumerated type and the other pointed-to type is the compatible integer type. C permits such operands to occur in the same *`conditional-expression`*. C++ does not. See pointer subtraction for an example. 1159 subtraction pointer operands

---

1272 — one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **`void`**.

C++

The C++ Standard does not support implicit conversions from pointer to **`void`** to pointers to other types (4.10p2). Therefore, this combination of operand types is not permitted.

```
int glob;
char *pc;
void *pv;

void f(void)
{
glob ? pc : pv; /* does not affect the conformance status of the program */
                // ill-formed
}
```

Semantics

1274 there is a sequence point after its evaluation. conditional operator sequence point

C++

5.16p1

All side effects of the first expression except for destruction of temporaries (12.2) happen before the second or third expression is evaluated.

function call sequence point 1025

The possible difference in behavior is the same as for the function-call operator.

---

conditional operator attempt to modify

If an attempt is made to modify the result of a conditional operator or to access it after the next sequence point, the behavior is undefined. 1278

C90

Wording to explicitly specify this undefined behavior is new in the C99 Standard.

C++

lvalue 721

The C++ definition of lvalue is the same as C90, so this wording is not necessary in C++.

Furthermore, if both operands are pointers to compatible types or to differently qualified versions of compatible types, the result type is a pointer to an appropriately qualified version of the composite type; 1283

C90

Furthermore, if both operands are pointers to compatible types or differently qualified versions of a compatible type, the result has the composite type;

The C90 wording did not specify that the appropriate qualifiers were added after forming the composite type. In:

```
extern int glob;
const enum {E1, E2} *p_ce;
volatile int *p_vi;

void f(void)
{
glob = *((p_e != p_i) ? p_vi : p_ce);
}
```

the pointed-to type, which is the composite type of the **enum** and **int** types, is also qualified with **const** and **volatile**.

---

otherwise, one operand is a pointer to **void** or a qualified version of **void**, in which case the result type is a pointer to an appropriately qualified version of **void**. 1285

C90

otherwise, one operand is a pointer to ***void*** *or a qualified version of* ***void****, in which case the other operand is converted to type pointer to* ***void****, and the result has that type.*

C90 did not add any qualifies to the pointer to **void** type. In the case of the **const** qualifier this difference would not have been noticeable (the resulting pointer type could not have been dereferenced without an explicit cast to modify the pointed-to object). In the case of the **volatile** qualifier this difference may result in values being accessed from registers in C90 while they will be accessed from storage in C99.

C++

The C++ Standard explicitly specifies the behavior for creating a composite pointer type (5.9p2) which is returned in this case.

1286 93) A conditional expression does not yield an lvalue. footnote 93

C++

If the second and third operands are lvalues and have the same type, the result is of that type and is an lvalue. 5.16p4

Otherwise, the result is an rvalue. 5.16p5

Source developed using a C++ translator may contain instances where the result of the conditional operator appears in an rvalue context, which will cause a constraint violation if processed by a C translator.

```
extern int glob;

void f(void)
{
short loc_s;
int loc_i;

((glob < 2) ? loc_i : glob) = 3;  /* constraint violation */
                                  // conforming
((glob > 2) ? loc_i : loc_s) = 3; // ill-formed
}
```

1287 EXAMPLE The common type that results when the second and third operands are pointers is determined in two independent stages. The appropriate qualifiers, for example, do not depend on whether the two pointers have compatible types. EXAMPLE ?: common pointer type

Given the declarations

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

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

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

C90

This example is new in C99.

6.5.16 Assignment operators

assignment-expression syntax

assignment-expression: 1288
    conditional-expression
    unary-expression assignment-operator assignment-expression
assignment-operator: one of

= *= /= %= += -= <<= >>= &= ^= |=

C++

5.17 *assignment-expression: conditional-expression logical-or-expression assignment-operator assignment-expression throw-expression*

footnote 85 1131 For some types, a cast is an lvalue in C++.

Constraints

assignment operator modifiable lvalue

An assignment operator shall have a modifiable lvalue as its left operand. 1289

C90

lvalue 721 The C99 Standard has removed the requirement, that was in C90, which lvalues refer to objects. This has resulted in the conformance status of the assignment `1=3` changing from a constraint violation to undefined behavior. The lvalue `1` does not designate an object and is not const-qualified. Therefore it is not ruled out from being modifiable in C99. lvalue 721

Semantics

assignment value of

An assignment expression has the value of the left operand after the assignment, but is not an lvalue. 1291

C++

5.17p1 *. . . ; the result is an lvalue.*

The C++ DR #222 (which at the time of this writing is at the drafting stage) queries some of the consequences of the result being an lvalue.

Source developed using a C++ translator may contain assignments that are a constraint violation if processed by a C translator.

```
extern int glob;

void f(void)
{
int x;
volatile int y;

(glob += 5) += 6; /* constraint violation */
                  // current status undefined behavior, object modified
```

```
                     // twice between sequence points.  The response to DR #222
                     // may add a sequence point, making the behavior defined

x = y = 0; /* equivalent to y=0; x=0; */
              // equivalent to y=0; x=y;
}
```

1293 The side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point. assignment when side effect occurs

C++

The C++ Standard does not explicitly state this requirement.

1294 The order of evaluation of the operands is unspecified. assignment operand evaluation order

C++

The C++ Standard does not explicitly make this observation.

1295 If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined.

C90

This sentence did not appear in the C90 Standard and had to be added to C99 because of a change in the definition of the term lvalue. 721 lvalue

C++

The C++ definition of lvalue is the same as C90, so this wording is not necessary in C++.

6.5.16.1 Simple assignment

Constraints

1296 One of the following shall hold:[94) simple assignment constraints

C++

The C++ Standard does not provide a list of constraints on the operands of any assignment operator (5.17). Clause 12.8 contains the specification that leads the following difference:

C1.8

The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

```
struct X { int i; };
struct X x1, x2;
volatile struct X x3 = {0};
x1 = x3;     // invalid C++
x2 = x3;     // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to volatile const X& would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.

— the left operand has qualified or unqualified arithmetic type and the right has arithmetic type; 1297

C++

5.17p3 *If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand.*

The conversions in clause 4 do not implicitly convert enumerated types to integer types and vice versa.

```
extern int glob;

enum {E1, E2};

void f(void)
{
glob = E1; /* does not affect the conformance status of the program */
           // ill-formed
}
```

assignment structure types

— the left operand has a qualified or unqualified version of a structure or union type compatible with the type of the right; 1298

C++

Clause 13.5.3 deals with this subject, but does not discuss this particular issue.

pointer qualified/unqualified versions

— both operands are pointers to qualified or unqualified versions of compatible types, and the type pointed to by the left has all the qualifiers of the type pointed to by the right; 1299

C++

The C++ wording (5.17p3) requires that an implicit conversion exist.

pointer 746 converting qualified/unqualified

The C++ requirements (4.4) on which implicit, qualified conversions are permitted are those described in the Smith paper (discussed elsewhere).

The pointer assignments supported by C++ are a superset of those supported by C. Source developed using a C++ translator may contain constraint violations if processed by a C translator, because it contains assignments between incompatible pointer types. The following example illustrates differences between the usages supported by C and C++ when types using two levels of pointer are declared.

```
void Jon_Krom(void)
{
/*
 * The issue of what is safe or unsafe is discussed elsewhere.
 * An example of case 3 is given in the standard.
 */

typedef int T; /* for any type T */

T       *       *  ppa  ;
T       *       *  ppb  ;

T       * const *  pcpa ;
T       * const *  pcpb ;

T const *       * cppa  ;
```

```
T const *       * cppb  ;

T const * const * cpcpa ;
T const * const * cpcpb ;

                  //  Safe       Allowed     Allowed
                  //   or         in          in
                  // Unsafe       C99         C++
                  // --------------------------------
ppb   = ppa ;     //  Safe        Yes         Yes      1
pcpb  = ppa ;     //  Safe        Yes         Yes      2
cppb  = ppa ;     //  Unsafe      No          No       3
cpcpb = ppa ;     //  Safe        No          Yes      4

ppb   = pcpa ;    //  Unsafe      No          No       5
pcpb  = pcpa ;    //  Safe        Yes         Yes      6
cppb  = pcpa ;    //  Unsafe      No          No       7
cpcpb = pcpa ;    //  Safe        No          Yes      8

ppb   = cppa ;    //  Unsafe      No          No       9
pcpb  = cppa ;    //  Unsafe      No          No      10
cppb  = cppa ;    //  Safe        Yes         Yes     11
cpcpb = cppa ;    //  Safe        Yes         Yes     12

ppb   = cpcpa ;   //  Unsafe      No          No      13
pcpb  = cpcpa ;   //  Unsafe      No          No      14
cppb  = cpcpa ;   //  Unsafe      No          No      15
cpcpb = cpcpa ;   //  Safe        Yes         Yes     16
}
```

1300 — one operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of **void**, and the type pointed to by the left has all the qualifiers of the type pointed to by the right;

C++

If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand. 5.17p3

The C++ Standard only supports an implicit conversion when the left operand has a pointer to **void** type, 4.10p2.

```
char *pc;
void *pv;

void f(void)
{
pc=pv; /* does not affect the conformance status of the program */
       // ill-formed
}
```

1302 or— the left operand has type **_Bool** and the right is a pointer.

C90

Support for the type **_Bool** is new in C99.

C++

Support for the type **_Bool** is new in C99 and is not specified in the C++ Standard. However, the C++ Standard does specify (4.12p1) that rvalues having pointer type can be converted to an rvalue of type **bool**.

Semantics

---

assignment value overlaps object

If the value being stored in an object is read from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have qualified or unqualified versions of a compatible type; 1304

C++

object types 475

The C++ Standard requires (5.18p8) that the objects have the same type. Even though the rvalue may have the same type as the lvalue (perhaps through the use of an explicit cast), this requirement is worded in terms of the object type. The C++ Standard is silent on the issue of overlapping objects.

```
enum E {E1, E2};
union {
       int m_1;
       enum E m_2;
      } x;

void f(void)
{
x.m_1 = (int)x.m_2; /* does not change the conformance status of the program */
                    // not defined?
}
```

---

footnote 94

94) The asymmetric appearance of these constraints with respect to type qualifiers is due to the conversion (specified in 6.3.2.1) that changes lvalues to "the value of the expression" ~~which~~and thus removes any type qualifiers ~~from the type category of the expression~~ that were applied to the type category of the expression (for example, it removes **const** but not **volatile** from the type **intvolatile*const**). 1307

C++

Even though the result of a C++ assignment operator is an lvalue, the right operand still needs to be converted to a value (except for reference types, but they are not in C) and the asymmetry also holds in C++.

6.5.16.2 Compound assignment

Constraints

For the other operators, each operand shall have arithmetic type consistent with those allowed by the corresponding binary operator. 1311

C++

5.15p7 *In all other cases, E1 shall have arithmetic type.*

Those cases where objects may not have some arithmetic type when appearing as operands to operators (i.e., floating types with the shift operators) are dealt with using the equivalence argument specified earlier in 5.15p7.

Semantics

6.5.17 Comma operator

Semantics

1315 there is a sequence point after its evaluation. comma operator sequence point

C++

> *All side effects (1.9) of the left expression, except for the destruction of temporaries (12.2), are performed before the evaluation of the right expression.* 5.18p1

The discussion on the function-call operator is applicable here. 1025 function call sequence point

1318 If an attempt is made to modify the result of a comma operator or to access it after the next sequence point, the behavior is undefined.

C90

This sentence did not appear in the C90 Standard and had to be added to C99 because of a change in the definition of the term lvalue. 721 lvalue

C++

The C++ definition of lvalue is the same as C90, so this wording is not necessary in C++. 721 lvalue

1320 95) A comma operator does not yield an lvalue. footnote 95 comma operator lvalue

C++

> *. . . ; the result is an lvalue if its right operand is.* 5.18p1

```
#include <stdio.h>

void DR_188(void)
{
char arr@lsquare[]100@rsquare[];

if (sizeof(0, arr) == sizeof(char *))
   printf("A C translator has been used\n");
else
   if (sizeof(0, arr) == sizeof(arr))
      printf("A C++ translator has been used\n");
   else
      printf("Who knows why we got here\n");
}

void f(void)
{
int loc;

(2, loc)=3; /* constraint violation */
            // conforming
}
```

6.6 Constant expressions

Description

A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be. 1323

C++

The C++ Standard says nothing about when constant expressions can be evaluated. It suggests (5,19p1) places where such constant expressions can be used. It has proved possible to write C++ source that require translators to calculate relatively complicated functions. The following example, from Veldhuizen,[5] implements the pow library function at translation time.

```
template<int I, int Y>
struct ctime_pow
   {
   static const int result = X * ctime_pow<X, Y-1>::result;
   };

// Base case to terminate recursion
template<int I>
struct ctime_pow<X, 0>
   {
   static const int results =- 1;
   };

const int x = ctime_pow<5, 3>::result; // assign five cubed to x
```

Constraints

constant expression not contain

Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.[96] 1324

C90

> *. . . they are contained within the operand of a* ***sizeof*** *operator.*[53]

sizeof result of

With the introduction of VLAs in C99 the result of the **sizeof** operator is no longer always a constant expression. The generalization of the wording to include any subexpression that is not evaluated means that nonconstant subexpressions can appear as operands to other operators (the logical-AND, logical-OR, and conditional operators). For instance, `0 || f()` can be treated as a constant expression. In C90 this expression, occurring in a context requiring a constant, would have been a constraint violation.

C++

Like C90, the C++ Standard only permits these operators to occur as the operand of a **sizeof** operator.
See C90 difference.

Each constant expression shall evaluate to a constant that is in the range of representable values for its type. 1325

C++

The C++ Standard does not explicitly specify an equivalent requirement.

Semantics

If a floating expression is evaluated in the translation environment, the arithmetic precision and range shall be at least as great as if the expression were being evaluated in the execution environment. 1327

C++

This requirement is not explicitly specified in the C++ Standard.

1328 An *integer constant expression*[97)] shall have integer type and shall only have operands that are integer constants, enumeration constants, character constants, **sizeof** expressions whose results are integer constants, and floating constants that are the immediate operands of casts. integer constant expression

C++

> *. . . , **const** variables or static data members of integral or enumeration types initialized with constant expressions (8.5), . . .* 5.19p1

For conforming C programs this additional case does not cause a change of behavior. But if a C++ translator is being used to develop programs that are intended to be conforming C, there is the possibility that this construct will be used.

```
const int ten = 10;

char arr@lsquare[]ten@rsquare[]; /* constraint violation */
                 // does not change the conformance status of the program
```

1330 More latitude is permitted for constant expressions in initializers.

C++

> *Other expressions are considered constant-expressions only for the purpose of non-local static object initialization (3.6.2).* 5.19p2

1334 96) The operand of a **sizeof** operator is usually not evaluated (6.5.3.4). footnote 96

C90

> *The operand of a **sizeof** operator is not evaluated (6.3.3.4) and thus any operator in 6.3 may be used.*

Unless the operand contains a VLA, which is new in C99, it will still not be evaluated.

C++

The operand is never evaluated in C++. This difference was needed in C99 because of the introduction of variable length array types.

1338 — an address constant for an object type plus or minus an integer constant expression.

C++

The C++ language requires that vendors provide a linker for a variety of reasons; for instance, support for name mangling.

1341 An *address constant* is a null pointer, a pointer to an lvalue designating an object of static storage duration, or a pointer to a function designator; address constant

C90

The C90 Standard did not explicitly state that the null pointer was an address constant, although all known implementations treated it as such.

C++

The C++ Standard does not include (5.19p4) a null pointer in the list of possible address constant expressions. Although a null pointer value is listed as being a *constant-expression* (5.19p2). This difference in terminology does not appear to result in any differences.

---

it shall be created explicitly using the unary & operator or an integer constant cast to pointer type, or implicitly by the use of an expression of array or function type. 1342

C90

Support for creating an address constant by casting an integer constant to a pointer type is new in C99. However, many C90 implementations supported this usage. It was specified as a future change by the response to DR #145.

C++

Like C90, the C++ Standard does not specify support for an address constant being created by casting an integer constant to a pointer type.

---

constant expression other forms

An implementation may accept other forms of constant expressions. 1344

C++

The C++ Standard does not given explicit latitude for an implementation to accept other forms of constant expression.

---

constant expression semantic rules

The semantic rules for the evaluation of a constant expression are the same as for nonconstant expressions.[98] 1345

C++

The C++ Standard does not explicitly state this requirement, in its semantics rules, for the evaluation of a constant expression.

---

footnote 98

98) Thus, in the following initialization, 1347

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

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

C++

The C++ Standard does not make this observation.

6.7 Declarations

declaration syntax

1348

```
declaration:
        declaration-specifiers init-declarator-list_opt ;
declaration-specifiers:
        storage-class-specifier declaration-specifiers_opt
        type-specifier declaration-specifiers_opt
        type-qualifier declaration-specifiers_opt
        function-specifier declaration-specifiers_opt
init-declarator-list:
        init-declarator
        init-declarator-list , init-declarator
```

init-declarator:
 declarator
 declarator = initializer

C90

Support for *function-specifier* is new in C99.

C++

The C++ syntax breaks declarations down into a number of different categories. However, these are not of consequence to C, since they involve constructs that are not available in C.

The nonterminal *type-qualifier* is called *cv-qualifier* in the C++ syntax. It also reduces through *type-specifier*, so the C++ abstract syntax tree is different from C. However, sequences of tokens corresponding to C declarations are accepted by the C++ syntax.

The C++ syntax specifies that *declaration-specifiers* is optional, which means that a semicolon could syntactically be interpreted as an empty declaration (it is always an empty statement in C). Other wording requires that one or the other always be specified. A source file that contains such a construct is not ill-formed and a diagnostic may not be produced by a translator.

Constraints

1349 A declaration shall declare at least a declarator (other than the parameters of a function or the members of a structure or union), a tag, or the members of an enumeration. declaration shall declare identifier

C90

A declaration shall declare at least a declarator, a tag, or the members of an enumeration. 6.5

The response to DR #155 pointed out that the behavior was undefined and that a diagnostic need not be issued for the examples below (which will cause a C99 implementation to issue a diagnostic).

```
struct { int mbr; }; /* Diagnostic might not appear in C90. */
union { int mbr; };  /* Diagnostic might not appear in C90. */
```

Such a usage is harmless in that it will not have affected the output of a program and can be removed by simple editing.

1350 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except for tags as specified in 6.7.2.3. declaration only one if no linkage

C++

C++ one definition rule

This requirement is called the *one definition rule* (3.2) in C++. There is no C++ requirement that a typedef name be unique within a given scope. Indeed 7.1.3p2 gives an explicit example where this is not the case (provided the redefinition refers to the type to which it already refers).

A program, written using only C constructs, could be acceptable to a conforming C++ implementation, but not be acceptable to a C implementation.

```
typedef int I;
typedef int I; // does not change the conformance status of the program
               /* constraint violation */
typedef I I;   // does not change the conformance status of the program
               /* constraint violation */
```

---

declarations refer to same object declarations refer to same function

All declarations in the same scope that refer to the same object or function shall specify compatible types. 1351

C++

3.5p10 *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4). A violation of this rule on type identity does not require a diagnostic.*

C++ requires identical types, while C only requires compatible types. A declaration of an object having an enumerated type and another declaration of the same identifier, using the compatible integer type, meets the C requirement but not the C++ one. However, a C++ translator is not required to issue a diagnostic if the declarations are not identical.

```
enum E {E1, E2};

extern enum E glob_E;
extern int glob_E; /* does not change the conformance status of the program */
                   // Undefined behavior, no diagnostic required

extern long glob_c;
extern long double glob_c; /* Constraint violation, issue a diagnostic message */
                           // Not required to       issue a diagnostic message
```

Semantics

---

definition identifier

A *definition* of an identifier is a declaration for that identifier that: 1353

C++

The C++ wording is phrased the opposite way around to that for C:

3.1p2 *A declaration is a definition unless . . .*

tentative definition 1849

The C++ Standard does not define the concept of tentative definition, which means that what are duplicate tentative definitions in C (a permitted usage) are duplicate definitions in C++ (an ill-formed usage).

```
int glob;     /* a tentative definition */
              // the definition

int glob = 5; /* the definition */
              // duplicate definition
```

1354 — for an object, causes storage to be reserved for that object; object reserve storage

C++

The C++ Standard does not specify what declarations are definitions, but rather what declarations are not definitions:

> *. . . , it contains the* **`extern`** *specifier (7.1.1) or a linkage-specification*[24] *(7.5) and neither an* `initializer` *nor a* `function-body`*, . . .* 3.1p2

> *An object declaration, however, is also a definition unless it contains the* **`extern`** *specifier and has no initializer (3.1).* 7p6

1355 — for a function, includes the function body;[99]

C++

The C++ Standard does not specify what declarations are definitions, but rather what declarations are not definitions:

> *A declaration is a definition unless it declares a function without specifying the function's body (8.4), . . .* 3.1p2

1356 — for an enumeration constant or typedef name, is the (only) declaration of the identifier.

C90

The C90 Standard did not specify that the declaration of these kinds of identifiers was also a definition, although wording in other parts of the document treated them as such. The C99 document corrected this defect (no formal DR exists). All existing C90 implementations known to your author treat these identifiers as definitions; consequently, no difference is specified here.

C++

> *A declaration is a definition unless . . . , or it is a* **`typedef`** *declaration (7.1.3), . . .* 3.1p2

A typedef name is not considered to be a definition in C++. However, this difference does not cause any C compatibility consequences. Enumerations constants are not explicitly excluded in the *unless* list. They are thus definitions.

1357 The declaration specifiers consist of a sequence of specifiers that indicate the linkage, storage duration, and part of the type of the entities that the declarators denote. declaration specifiers

C++

The C++ Standard does not make this observation.

---

declarator list of

The init-declarator-list is a comma-separated sequence of declarators, each of which may have additional type information, or an initializer, or both. 1358

C++

The C++ Standard does not make this observation.

---

object type complete by end

If an identifier for an object is declared with no linkage, the type for the object shall be complete by the end of its declarator, or by the end of its init-declarator if it has an initializer; 1361

C++

3.1p6 *A program is ill-formed if the definition of any object gives the object an incomplete type (3.9).*

The C++ wording covers all of the cases covered by the C specification above.

A violation of this requirement must be diagnosed by a conforming C++ translator. There is no such requirement on a C translator. However, it is very unlikely that a C implementation will not issue a diagnostic in this case (perhaps because of some extension being available).

---

in the case of function ~~arguments~~ parameters (including in prototypes), it is the adjusted type (see 6.7.5.3) that is required to be complete. 1362

C90

This wording was added to the C99 Standard to clarify possible ambiguities in the order in which requirements, in the standard, were performed on parameters that were part of a function declaration; for instance, `int f(int a[]);`.

C++

The nearest the C++ Standard comes to specifying such a rule is:

5.2.2p4 *When a function is called, the parameters that have object type shall have completely-defined object type. [Note: this still allows a parameter to be a pointer or reference to an incomplete class type. However, it prevents a passed-by-value parameter to have an incomplete class type.]*

6.7.1 Storage-class specifiers

storage-class specifier syntax

--- 1364

```
storage-class-specifier:
        typedef
        extern
        static
        auto
        register
```

C++

The C++ Standard classifies **typedef** (7.1p1) as a *decl-specifier*, not a *storage-class-specifier* (which also includes **mutable**, a C++ specific keyword).

Constraints

1365 At most, one storage-class specifier may be given in the declaration specifiers in a declaration.[100)]

C++

While the C++ Standard (7.1.1p1) contains the same requirement, it does not include **typedef** in the list of *storage-class-specifiers*. There is no wording in the C++ limiting the number of instances of the **typedef** *decl-specifier* in a declaration.

Source developed using a C++ translator may contain more than one occurrence of the **typedef** *decl-specifier* in a declaration.

Semantics

1366 The **typedef** specifier is called a "storage-class specifier" for syntactic convenience only;

C++

It is called a *decl-specifier* in the C++ Standard (7.1p1).

1369 A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. register storage-class

C++

> *A **register** specifier has the same semantics as an **auto** specifier together with a hint to the implementation that the object so declared will be heavily used.* 7.1.1p3

Translator implementors are likely to assume that the reason a developer provides this hint is that they are expecting the translator to make use of it to improve the performance of the generated machine code. The C++ hint does not specify implementation details. The differing interpretations given, by the two standards, for hints provides to translators is not likely to be significant. The majority of modern translators ignore the hint and do what they think is best.

1370 The extent to which such suggestions are effective is implementation-defined.[101)] register extent effective

C++

The C++ Standard gives no status to a translator's implementation of this hint (suggestion). A C++ translator is not required to document its handling of the **register** storage-class specifier and often a developer is no less wiser than if it is documented.

1374 However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier **register** cannot be computed, either explicitly (by use of the unary **&** operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1).

C++

This requirement does not apply in C++. 1088 unary & operand constraints

1375 Thus, the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof**.

C++

This observation is not true in C++. 1088 unary & operand constraints

1376 If an aggregate or union object is declared with a storage-class specifier other than **typedef**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects.

C90

This wording did not appear in the C90 Standard and was added by the response to DR #017q6.

C++

The C++ Standard does not explicitly specify the behavior in this case.

6.7.2 Type specifiers

type specifier syntax

1378

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

C90

Support for the *type-specifiers* **_Bool**, **_Complex**, and **_Imaginary** is new in C99.

C++

The nonterminal for these terminals is called *simple-type-specifier* in C++ (7.1.5.2p1). The C++ Standard does contain a nonterminal called *type-specifier*. It is used in a higher-level production (7.1.5p1) that includes *cv-qualifier*.

The C++ Standard includes **wchar_t** and **bool** (the identifier bool is defined as a macro in the header **stdbool.h** in C) as *type-specifiers* (they are keywords in C++). The C++ Standard does not include **_Bool**, **_Complex** and **_Imaginary**, either as keywords or type specifiers.

Constraints

declaration at least one type specifier

At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each struct declaration and type name. 1379

C90

This requirement is new in C99.

In C90 an omitted *type-specifier* implied the type specifier **int**. Translating a source file that contains such a declaration will cause a diagnostic to be issued and are no longer considered conforming programs.

C++

7.1.5p2 *At least one* type-specifier *that is not a* cv-qualifier *is required in a declaration unless it declares a constructor, destructor or conversion function.*[80)]

Although the terms used have different definitions in C/C++, the result is the same.

type specifiers possible sets of

1382
--~ **void**

--~ **char**
--~ **signed char**
--~ **unsigned char**
--~ **short**, **signed short**, **short int**, or **signed short int**
--~ **unsigned short**, or **unsigned short int**
--~ **int**, **signed**, or **signed int**
--~ **unsigned**, or **unsigned int**
--~ **long**, **signed long**, **long int**, or **signed long int**
--~ **unsigned long**, or **unsigned long int**
--~ **long long**, **signed long long**, **long long int**, or **signed long long int**
--~ **unsigned long long**, or **unsigned long long int**
--~ **float**
--~ **double**
--~ **long double**
--~ **_Bool**
--~ **float _Complex**
--~ **double _Complex**
--~ **long double _Complex**
~~--~**float _Imaginary**~~
~~--~**double _Imaginary**~~
~~--~**long double _Imaginary**~~
--~ struct or union specifier
--~ enum specifier
--~ typedef name

C90

Support for the following is new in C99:

— **long long**, **signed long long**, **long long int**, or **signed long long int**
- — **unsigned long long**, or **unsigned long long int**
- — **_Bool**
- — **float _Complex**
- — **double _Complex**
- — **long double _Complex**

Support for the *no type specifiers* set, in the **int**, **signed**, **signed int** list has been removed in C99.

```
extern x;   /* strictly conforming C90  */
            /* constraint violation C99 */
const  y;   /* strictly conforming C90  */
            /* constraint violation C99 */
       z;   /* strictly conforming C90  */
            /* constraint violation C99 */
       f(); /* strictly conforming C90  */
            /* constraint violation C99 */
```

C++

The list of combinations, given above as being new in C99 are not supported by C++.

Like C99, the C++ Standard does not require a translator to provide an implicit function declaration returning **int** (footnote 80) being supplied for a missing type specifier.

The type specifier~~s~~ ~~**_Complex** and **_Imaginary**~~ shall not be used if the implementation does not provide ~~those~~ complex types.[102)] 1383

C90

Support for this type specifier is new in C99.

C++

Support for these type specifiers is new in C99 and are not specified as such in the C++ Standard. The header **<complex>** defines template classes and associated operations whose behavior provides the same functionality as that provided, in C, for objects declared to have type **_Complex**. There are no equivalent definitions for **_Imaginary**.

Semantics

---

bit-field int

Each of the comma-separated sets designates the same type, except that for bit-fields, it is implementation-defined whether the specifier **int** designates the same type as **signed int** or the same type as **unsigned int**. 1387

C90

> *Each of the above comma-separated sets designates the same type, except that for bit-fields, the type **signed int** (or **signed**) may differ from **int** (or no type specifiers).*

C++

Rather than giving a set of possibilities, the C++ Standard lists each combination of specifiers and its associated type (Table 7).

---

footnote 102

102) ~~101)Implementations are not required to provide imaginary types.~~ Freestanding implementations are not required to provide complex types. 1388

C90

Support for complex types is new in C99.

C++

There is no specification for imaginary types (in the **<complex>** header or otherwise) in the C++ Standard.

6.7.2.1 Structure and union specifiers

struct/union syntax

1390

```
struct-or-union-specifier:
            struct-or-union identifier_opt { struct-declaration-list }
            struct-or-union identifier
struct-or-union:
            struct
            union
struct-declaration-list:
```

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

        specifier-qualifier-list struct-declarator-list ;
specifier-qualifier-list:
        type-specifier specifier-qualifier-list_opt
        type-qualifier specifier-qualifier-list_opt
struct-declarator-list:
        struct-declarator
        struct-declarator-list , struct-declarator
struct-declarator:
        declarator
        declarator_opt : constant-expression
```

C++

The C++ Standard uses the general term *class* to refer to these constructs. This usage is also reflected in naming of the nonterminals in the C++ syntax. The production *struct-or-union* is known as *class-key* in C++ and also includes the keyword **class**. The form that omits the brace enclosed list of members is known as an *elaborated-type-specifier* (7.1.5.3) in C++.

Constraints

1391 A structure or union shall not contain a member with incomplete or function type (hence, a structure shall not contain an instance of itself, but may contain a pointer to an instance of itself), except that the last member of a structure with more than one named member may have incomplete array type; member not types

C90

Support for the exception on the last named member is new in C99.

C++

It is a design feature of C++ that class types can contain incomplete and function types. Source containing instances of such constructs is making use of significant features of C++ and there is unlikely to be any expectation of being able to successfully process it using a C translator.

The exception on the last named member is new in C99 and this usage is not supported in the C++ Standard.

The following describes a restriction in C++ that does not apply in C.

annex C.1.7p3

Change: *In C++, a* **typedef** *name may not be redefined in a class declaration after being used in the declaration*

Example:

```
typedef int I;
struct S {
         I i;
         int I;  // valid C, invalid C++
        };
```

> ***Rationale:*** *When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.*

---

bit-field maximum width

The expression that specifies the width of a bit-field shall be an integer constant expression that has a nonnegative value that shall not exceed the ~~number~~width of ~~bits in~~ an object of the type that ~~is~~would be specified ~~if~~were the colon and expression ~~are~~ omitted. 1393

C90

The C90 wording ended with " . . . of bits in an ordinary object of compatible type.", which begs the question of whether bit-fields are variants of integer types or are separate types.

C++

bit-field 575 value is m bits

The C++ issues are discussed elsewhere.

If the value is zero, the declaration shall have no declarator. 1394

C++

9.6p2

> *Only when declaring an unnamed bit-field may the `constant-expression` be a value equal to zero.*

Source developed using a C++ translator may contain a declaration of a zero width bit-field that include a declarator, which will generate a constraint violation if processed by a C translator.

```
struct {
        int mem_1;
        unsigned int mem_2:0; //                    no diagnostic required
                              /* constraint violation, diagnostic required */
       } obj;
```

There is an open C++ DR (#057) concerning the lack of a prohibition against declarations of the form:

```
union {int : 0;} x;
```

---

bit-field shall have type

A bit-field shall have a type that is a qualified or unqualified version of **`_Bool`**, **`signed int`**, **`unsigned int`**, or some other implementation-defined type. 1395

C90

The following wording appeared in a semantics clause in C90, not a constraint clause.

> *A bit-field shall have a type that is a qualified or unqualified version of one of* ***`int`***, ***`unsigned int`***, *or* ***`signed int`***.

Programs that used other types in the declaration of a bit-field exhibited undefined behavior in C90. Such programs exhibit implementation-defined behavior in C99.

C++

9.6p3

A bit-field shall have integral or enumeration type (3.9.1).

Source developed using a C++ translator may contain bit-fields declared using types that are a constraint violation if processed by a C translator.

```
enum E_TAG {a, b};

struct {
        char m_1  : 3;
        short m_2 : 5;
        long m_3  : 7;
        enum E_TAG m_4 : 9;
       } glob;
```

Semantics

1396 As discussed in 6.2.5, a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence, and a union is a type consisting of a sequence of members whose storage overlap.

C++

This requirement can be deduced from 9.2p12 and 9.5p1.

1397 Structure and union specifiers have the same form.

C++

The C++ Standard does not make this observation.

1401 If the struct-declaration-list contains no named members, the behavior is undefined.

C++

An object of a class consists of a (possibly empty) sequence of members and base class objects. 9p1

Source developed using a C++ translator may contain class types having no members. This usage will result in undefined behavior when processed by a C translator.

1403 A member of a structure or union may have any object type other than a variably modified type.[103)] struct member type

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and they are not specified in the C++ Standard.

1407 A bit-field is interpreted as a signed or unsigned integer type consisting of the specified number of bits.[105)] bit-field interpreted as

C++

The C++ Standard does not specify (9.6p1) that the specified number of bits is used for the value representation.

1408 If the value 0 or 1 is stored into a nonzero-width bit-field of type **_Bool**, the value of the bit-field shall compare equal to the value stored.

C90

Support for the type `_Bool` is new in C99.

---

bit-field packed into

If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. 1410

C++

This requirement is not specified in the C++ Standard.

9.6p1 *Allocation of bit-fields within a class object is implementation-defined.*

---

alignment addressable storage unit

The alignment of the addressable storage unit is unspecified. 1413

C++

The wording in the C++ Standard refers to the bit-field, not the addressable allocation unit in which it resides. Does this wording refer to the alignment within the addressable allocation unit?

9.6p1 *Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit.*

---

footnote 103

103) A structure or union can not contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3. 1416

C90

Support for variably modified types is new in C99.

C++

Variably modified types are new in C99 and are not available in C++.

---

footnote 105

105) As specified in 6.7.2 above, if the actual type specifier used is `int` or a typedef-name defined as `int`, then it is implementation-defined whether the bit-field is signed or unsigned. 1419

C90

This footnote is new in C99.

---

member alignment

Each non-bit-field member of a structure or union object is aligned in an implementation-defined manner appropriate to its type. 1421

C++

The C++ Standard specifies (3.9p5) that the alignment of all object types is implementation-defined.

---

member address increasing

Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared. 1422

C++

The C++ Standard does not say anything explicit about bit-fields (9.2p12).

---

structure unnamed padding

There may be unnamed padding within a structure object, but not at its beginning. 1424

C90

There may therefore be unnamed padding within a structure object, but not at its beginning, as necessary to achieve the appropriate alignment.

C++

This commentary applies to POD-struct types (9.2p17) in C++. Such types correspond to the structure types available in C.

1427 A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa. union members start same address

C++

This requirement can be deduced from:

Each data member is allocated as if it were the sole member of a **`struct`**. 9.5p1

1428 There may be unnamed padding at the end of a structure or union. structure trailing padding

C++

The only time this possibility is mentioned in the C++ Standard is under the **`sizeof`** operator:

When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array. 5.3.3p2

1429 As a special case, the last element of a structure with more than one named member may have an incomplete array type;

C90

The issues involved in making use of the *struct hack* were raised in DR #051. The response pointed out declaring the member to be an array containing fewer elements and then allocating storage extra storage for additional elements was not strictly conforming. However, declaring the array to have a large number of elements and allocating storage for fewer elements was strictly conforming.

```
#include <stdlib.h>
#define HUGE_ARR 10000 /* Largest desired array. */

struct A {
          char x@lsquare[]HUGE_ARR@rsquare[];
         };

int main(void)
{
struct A *p = (struct A *)malloc(sizeof(struct A)
                                  - HUGE_ARR + 100); /* Want x@lsquare[]100@rsquare[] this time. */
p->x@lsquare[]5@rsquare[] = '?'; /* Is strictly conforming. */
return 0;
}
```

Support for the last member having an incomplete array type is new in C99.

C++

Support for the last member having an incomplete array type is new in C99 and is not available in C++.

---

flexible array member

this is called a *flexible array member*. 1430

C++

There is no equivalent construct in C++.

6.7.2.2 Enumeration specifiers

---

enumeration specifier syntax

1439

```
enum-specifier:
        enum identifier_opt { enumerator-list }
        enum identifier_opt { enumerator-list , }
        enum identifier
enumerator-list:
        enumerator
        enumerator-list , enumerator
enumerator:
        enumeration-constant
        enumeration-constant = constant-expression
```

C90

Support for a trailing comma at the end of an `enumerator-list` is new in C99.

C++

The form that omits the brace enclosed list of members is known as an elaborated type specifier, 7.1.5.3, in C++.

The C++ syntax, 7.2p1, does not permit a trailing comma.

Constraints

---

enumeration constant representable in int

The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an **`int`**. 1440

C++

7.2p1 *The `constant-expression` shall be of integral or enumeration type.*

7.2p4 *If an initializer is specified for an enumerator, the initializing value has the same type as the expression.*

Source developed using a C++ translator may contain enumeration initialization values that would be a constraint violation if processed by a C translator.

```
#include <limits.h>

enum { umax_int = UINT_MAX}; /* constraint violation */
                             // has type unsigned int
```

Semantics

1441 The identifiers in an enumerator list are declared as constants that have type **int** and may appear wherever such are permitted.[107)] enumerators type int

C++

> *Following the closing brace of an* `enum-specifier`*, each enumerator has the type of its enumeration. Prior to the closing brace, the type of each enumerator is the type of its initializing value.* 7.2p4

In C the type of an enumeration constant is always **int**, independently of the integer type that is compatible with its enumeration type.

```
#include <limits.h>

int might_be_cpp_translator(void)
{
enum { a = -1, b = UINT_MAX }; // each enumerator fits in int or unsigned int

return (sizeof(a) != sizeof(int));
}

void CPP_DR_172_OPEN(void) // Open C++ DR
{
enum { zero };

if (-1 < zero) /* always true */
               // might be false (because zero has an unsigned type)
   ;
}
```

1445 (The use of enumerators with = may produce enumeration constants with values that duplicate other values in the same enumeration.)

C++

The C++ Standard does not explicitly mention this possibility, although it does give an example, 7.2p2, of an enumeration type containing more than one enumeration constant having the same value.

1446 The enumerators of an enumeration are also known as its members.

C++

The C++ Standard does not define this additional terminology for enumerators; probably because it is strongly associated with a different meaning for members of a class.

> *. . . the associated* `enumerator` *the value indicated by the* `constant-expression`*.* 7.2p1

1447 Each enumerated type shall be compatible with **char**, a signed integer type, or an unsigned integer type. enumeration type compatible with

C90

Each enumerated type shall be compatible with an integer type;

integer types 519

The integer types include the enumeration types. The change of wording in the C99 Standard removes a circularity in the specification.

C++

7.2p1 *An enumeration is a distinct type (3.9.1) with named constants.*

enumeration constant type 864
enumeration different type 518

The underlying type of an enumeration may be an integral type that can represent all the enumerator values defined in the enumeration (7.2p5). But from the point of view of type compatibility it is a distinct type.

7.2p5 *It is implementation-defined which integral type is used as the underlying type for an enumeration except that the underlying type shall not be larger than **int** unless the value of an enumerator cannot fit in an **int** or **unsigned int**.*

While it is possible that source developed using a C++ translator may select a different integer type than a particular C translator, there is no effective difference in behavior because different C translators may also select different types.

The choice of type is implementation-defined,[108)] but shall be capable of representing the values of all the members of the enumeration. 1448

C90

The requirement that the type be capable of representing the values of all the members of the enumeration was added by the response to DR #071.

---

enumerated type incomplete until

The enumerated type is incomplete until after the **}** that terminates the list of enumerator declarations. 1449

C90

The C90 Standard did not specify when an enumerated type was completed.

C++

The C++ Standard neither specifies that the enumerated type is incomplete at any point or that it becomes complete at any point.

7.2p4 *Following the closing brace of an* `enum-specifier`*, each enumerator has the type of its enumeration*

EXAMPLE The following fragment: 1450

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

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

C++

The equivalent example in the C++ Standard uses the enumeration names red, yellow, green and blue.

1452 107) Thus, the identifiers of enumeration constants declared in the same scope shall all be distinct from each other and from other identifiers declared in ordinary declarators. footnote 107

C++

The C++ Standard does not explicitly make this observation.

1453 108) An implementation may delay the choice of which integer type until all enumeration constants have been seen. footnote 108

C90

The C90 Standard did not make this observation about implementation behavior.

C++

This behavior is required of a C++ implementation because:

> *The underlying type of an enumeration is an integral type that can represent all the enumerator values defined in the enumeration.* 7.2p5

6.7.2.3 Tags

1454 A specific type shall have its content defined at most once. type contents defined once

C90

This requirement was not explicitly specified in the C90 Standard (although it might be inferred from the wording), but was added by the response to DR #165.

C++

The C++ Standard does not classify the identifiers that occur after the **enum**, **struct**, or **union** keywords as tags. There is no tag namespace. The identifiers exist in the same namespace as object and typedef identifiers. This namespace does not support multiple definitions of the same name in the same scope (3.3p4). It is this C++ requirement that enforces the C one given above.

1455 Where two declarations that use the same tag declare the same type, they shall both use the same choice of **struct**, **union**, or **enum**. tag name same struct, union or enum

C90

The C90 Standard did not explicitly specify this constraint. While the behavior was therefore undefined, it is unlikely that the behavior of any existing code will change when processed by a C99 translator (and no difference is flagged here).

C++

> *The `class-key` or **`enum`** keyword present in the `elaborated-type-specifier` shall agree in kind with the declaration to which the name in the `elaborated-type-specifier` refers.* 7.1.5.3p3

1456 A type specifier of the form

```
enum identifier
```

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

C90

This C99 requirement was not specified in C90, which did not containing any wording that ruled out the declaration of an incomplete enumerated type (and confirmed by the response to DR #118). Adding this constraint brings the behavior of enumeration types in line with that for structure and union types.

sizeof constraints 1118

Source code containing declarations of incomplete enumerator types will cause C99 translators to issue a diagnostic, where a C90 translator was not required to issue one.

```
enum E1 { ec_1 = sizeof (enum E1) };   /* Constraint violation in C99. */
enum E2 { ec_2 = sizeof (enum E2 *) }; /* Constraint violation in C99. */
```

C++

3.3.1p5 *[Note: if the* `elaborated-type-specifier` *designates an enumeration, the* `identifier` *must refer to an already declared* `enum-name`*.*

3.4.4p2 *If the* `elaborated-type-specifier` *refers to an* `enum-name` *and this lookup does not find a previously declared* `enum-name`*, the* `elaborated-type-specifier` *is ill-formed.*

```
enum incomplete_tag *x; /* constraint violation */
                        // undefined behavior

enum also_incomplete; /* constraint violation */
                      // ill-formed
```

Semantics

tag declarations same scope

All declarations of structure, union, or enumerated types that have the same scope and use the same tag declare the same type. 1457

C90

This requirement was not explicitly specified in the C90 Standard (although it might be inferred from the wording), but was added by the response to DR #165.

C++

The C++ Standard specifies this behavior for class types (9.1p2). While this behavior is not specified for enumerated types, it is not possible to have multiple declarations of such types.

tag incomplete until

The type is incomplete[109)] until the closing brace of the list defining the content, and complete thereafter. 1458

C++

The C++ Standard specifies this behavior for class types (9.2p2), but is silent on the topic for enumerated types.

tag declarations different scope

Two declarations of structure, union, or enumerated types which are in different scopes or use different tags declare distinct types. 1459

C90

The C99 Standard more clearly specifies the intended behavior, which had to be inferred in the C90 Standard. 1457 tag declarations same scope

C++

The C++ Standard specifies this behavior for class definitions (9.1p1), but does not explicitly specify this behavior for declarations in different scope.

1460 Each declaration of a structure, union, or enumerated type which does not include a tag declares a distinct type. struct/union declaration no tag

C90

The C90 Standard refers to a " . . . a new structure, union, or enumerated type," without specifying the distinctness of new types. The C99 Standard clarified the meaning.

C++

The C++ Standard specifies that a class definition introduces a new type, 9.1p1 (which by implication is distinct). However, it does not explicitly specify the status of the type that is created when the tag (a C term) is omitted in a declaration.

1461 A type specifier of the form

struct-or-union identifier$_{opt}$ **{** *struct-declaration-list* **}**

or

enum *identifier* **{** *enumerator-list* **}**

or

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

declares a structure, union, or enumerated type.

C90

Support for the comma terminated form of enumerated type declaration is new in C99.

C++

The C++ Standard does not explicitly specify this semantics (although 9p4 comes close).

1463 If an identifier is provided,[110)] the type specifier also declares the identifier to be the tag of that type. tag declare

C++

The term *tag* is not used in C++, which calls the equivalent construct a *class name*.

1465 109) An incomplete type may only by used when the size of an object of that type is not needed. footnote 109 size needed

C90

> *It declares a tag that specifies a type that may be used only when the size of an object of the specified type is not needed.*

The above sentence appears in the main body of the standard, not a footnote.

The C99 wording is more general in that it includes all incomplete types. This is not a difference in behavior because these types are already allowed to occur in the same context as an incomplete structure/union type. 475 incomplete types

C++

The C++ Standard contains no such rule, but enumerates the cases:

3.9p8 *[Note: the rules for declarations and expressions describe in which contexts incomplete types are prohibited.]*

The specification has to be complete before such a function is called or defined. 1467

C90

The specification shall be complete before such a function is called or defined.

The form of wording has been changed from appearing to be a requirement (which would not be normative in a footnote) to being commentary.

---

footnote 110

110) If there is no identifier, the type can, within the translation unit, only be referred to by the declaration of which it is a part. 1468

C90

This observation was is new in the C90 Standard.

C++

The C++ Standard does not make this observation.

Of course, when the declaration is of a typedef name, subsequent declarations can make use of that typedef name to declare objects having the specified structure, union, or enumerated type. 1469

C90

This observation is new in the C90 Standard.

C++

The C++ Standard does not make this observation.

---

footnote 111

111) A similar construction with **`enum`** does not exist. 1470

C++

7.1.5.3p1 *If an `elaborated-type-specifier` is the sole constituent of a declaration, the declaration is ill-formed unless . . .*

struct-or-union identifier not visible 1471

The C++ Standard does not list `enum identifier ;` among the list of exceptions and a conforming C++ translator is required to issue a diagnostic for any instances of this usage.

The C++ Standard agrees with this footnote for its second reference in the C90 Standard.

---

struct-or-union identifier not visible

If a type specifier of the form 1471

`struct-or-union identifier`

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

C++

The C++ Standard does not explicitly discuss this kind of construction/occurrence, although 3.9p6 and 3.9p7 discuss this form of incomplete type.

1472 If a type specifier of the form struct-or-union identifier visible

struct-or-union identifier

or

enum *identifier*

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

C++

3.4.4p2 covers this case .E_COMMENT

1474 EXAMPLE 2 To illustrate the use of prior declaration of a tag to specify a pair of mutually referential structures, the declarations EXAMPLE mutually referential structures

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

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

```
struct s2;
```

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

C++

This form of declaration would not have the desired affect in C++ because the braces form a scope. The declaration of s2 would need to be completed within that scope, unless there was a prior visible declaration it could refer to.

6.7.3 Type qualifiers

1476 type qualifier syntax

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

C90

Support for **restrict** is new in C99.

C++

Support for **restrict** is new in C99 and is not specified in the C++ Standard.

Constraints

Semantics

1478 The properties associated with qualified types are meaningful only for expressions that are lvalues.[112)] qualifier meaningful for lvalues

C++

The C++ Standard also associates properties of qualified types with rvalues (3.10p3). Such cases apply to constructs that are C++ specific and not available in C.

---

qualifier appears more than once

If the same qualifier appears more than once in the same *specifier-qualifier-list*, either directly or via one or more **typedef**s, the behavior is the same as if it appeared only once. 1479

C90

The following occurs within a Constraints clause.

> *The same type qualifier shall not appear more than once in the same specifier list or qualifier list, either directly or via one or more* ***typedef****s.*

Source code containing a declaration with the same qualifier appearing more than once in the same *specifier-qualifier-list* will cause a C90 translator to issue a diagnostic.

C++

7.1.5p1

> *However, redundant cv-qualifiers are prohibited except when introduced through the use of typedefs (7.1.3) or template type arguments (14.3), in which case the redundant cv-qualifiers are ignored.*

The C++ Standard does not define the term *prohibited*. Applying common usage to this term suggests that it is to be interpreted as a violation of a diagnosable (because "no diagnostic is required", 1.4p1, has not been specified) rule.

The C++ specification is intermediate between that of C90 and C99.

```
const const int cci; /* does not change the conformance status of program */
                     // in violation of a diagnosable rule
```

---

const qualified attempt modify

If an attempt is made to modify an object defined with a const-qualified type through use of an lvalue with non-const-qualified type, the behavior is undefined. 1480

C++

3.10p10

> *An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances. [Example: a member function called for an object (9.3) can modify the object.]*

static internal linkage 425

The C++ Standard specifies a different linkage for some objects declared using a const-qualified type.

Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. 1483

C++

There is no equivalent statement in the C++ Standard. But it can be deduced from the following two paragraphs:

1.9p5

> *A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible execution sequences of the corresponding instance of the abstract machine with the same program and the same input.*

> *The observable behavior of the abstract machine is its sequence of reads and writes to* **`volatile`** *data and calls to library I/O functions.*[6)] 1.9p6

1485 112) The implementation may place a **`const`** object that is not **`volatile`** in a read-only region of storage. footnote 112

C++

The C++ Standard does not make this observation.

1486 Moreover, the implementation need not allocate storage for such an object if its address is never used.

C++

The C++ Standard does not make this observation. However, given C++ supports zero sized objects, 1.8p5, there may be other cases where implementations need not allocate storage.

1487 113) This applies to those objects that behave as if they were defined with qualified types, even if they are never actually defined as objects in the program (such as an object at a memory-mapped input/output address). footnote 113

C++

The C++ Standard does not make this observation.

1488 What constitutes an access to an object that has volatile-qualified type is implementation-defined.

C++

The C++ Standard does not explicitly specify any behavior.

> *[Note: . . . In general, the semantics of* **`volatile`** *are intended to be the same in C++ as they are in C.]* 7.1.5.1p8

1489 An object that is accessed through a restrict-qualified pointer has a special association with that pointer.

C90

Support for the **`restrict`** qualifier is new in C99.

C++

Support for the **`restrict`** qualifier is new in C99 and is not available in C++.

1493 If the specification of a function type includes any type qualifiers, the behavior is undefined.[116)]

C++

8.3.5p4

In fact, if at any time in the determination of a type a cv-qualified function type is formed, the program is ill-formed.

A C++ translator will issue a diagnostic for the appearance of a qualifier in a function type, while a C translator may silently ignore it.

The C++ Standard allows *cv-qualifiers* to appear in a function declarator. The syntax is:

8.3.5p1 *D1 (parameter-declaration-clause) cv-qualifier-seq$_{opt}$ exception-specification$_{opt}$*

the *cv-qualifier* occurs after what C would consider to be the function type.

---

qualified type to be compatible

For two qualified types to be compatible, both shall have the identically qualified version of a compatible type; 1494

compatible type if 631

C++

The C++ Standard does not define the term *compatible type*. However, the C++ Standard does define the terms *layout-compatible* (3.9p11) and *reference-compatible* (8.5.3p4). However, *cv-qualifiers* are not included in the definition of these terms.

---

the order of type qualifiers within a list of specifiers or qualifiers does not affect the specified type. 1495

C++

The C++ Standard does not specify any ordering dependency on *cv-qualifiers* within a *decl-specifier*.

---

footnote 114

114) A **volatile** declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. 1498

C++

The C++ Standard does not make this observation.

Actions on objects so declared shall not be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions. 1499

C++

7.1.5.1p8 *[Note: **volatile** is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. See 1.9 for detailed semantics. In general, the semantics of **volatile** are intended to be the same in C++ as they are in C.]*

---

footnote 116

116) Both of these can occur through the use of **typedef**s. 1501

C++

The C++ Standard does not make this observation, but it does include the following example:

8.3.5p4 *[Example:*

```
typedef void F();
struct S {
const F f;     // ill-formed:
               // not equivalent to: void f() const;
};
```

—end example]

6.7.3.1 Formal definition of restrict

1502 Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a restrict-qualified pointer to type **T**. restrict formal definition

C++

Support for the **restrict** qualifier is new in C99 and is not available in C++.

6.7.4 Function specifiers

function specifier syntax

1522

```
function-specifier:
        inline
```

C90

Support for *function-specifier* is new in C99.

C++

The C++ Standard also includes, 7.1.2p1, the *function-specifier*s **virtual** and **explicit**.

Constraints

1524 An inline definition of a function with external linkage shall not contain a definition of a modifiable object with static storage duration, and shall not contain a reference to an identifier with internal linkage. inline static storage duration

C++

The C++ Standard does not contain an equivalent prohibition.

> *A **static** local variable in an **extern inline** function always refers to the same object.* 7.1.2p4

The C++ Standard does not specify any requirements involving a static local variable in a static inline function. Source developed using a C++ translator may contain inline function definitions that would cause a constraint violation if processed by a C translator.

1525 In a hosted environment, the **inline** function specifier shall not appear in a declaration of **main.**

C++

> *A program that declares `main` to be **inline** or **static** is ill-formed.* 3.6.1p3

A program, in a freestanding environment, which includes a declaration of the function `main` (which need not exist in such an environment) using the **inline** function specifier will result in a diagnostic being issued by a C++ translator.

Semantics

1527 The function specifier may appear more than once; function specifier appears more than once

C++

The C++ Standard does not explicitly specify this case (which is supported by its syntax).

1529 Making a function an inline function suggests that calls to the function be as fast as possible.[118)] inline suggests fast calls

C++

The C++ Standard gives an implementation technique, not a suggestion of intent:

7.1.2p2 *The **`inline`** specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism.*

Such wording does not prevent C++ implementors interpreting the function specifier in the C Standard sense (by, for instance, giving instructions to the hardware memory manager to preferentially keep a function's translated machine code in cache).

The extent to which such suggestions are effective is implementation-defined.[119] 1530

C++

7.1.2p2 *An implementation is not required to perform this inline substitution at the point of call;*

A C++ implementation is not required to document its behavior.

Any function with internal linkage can be an inline function. 1531

C++

The C++ Standard does not explicitly give this permission (any function declaration can include the **`inline`** specifier, but this need not have any effect).

For a function with external linkage, the following restrictions apply: 1532

C++

The C++ Standard also has restrictions on inline functions having external linkage. But it does not list them in one paragraph.

A program built from the following source files is conforming C, but is ill-formed C++ (3.2p5).

File a.c
```
inline int f(void)
{
return 0+0;
}
```

File b.c
```
int f(void)
{
return 0;
}
```

Building a program from sources files that have been translated by different C translators requires that various external interface issues, at the object code level, be compatible. The situation is more complicated when the translated output comes from both a C and a C++ translator. The following is an example of a technique that might be used to handle some inline functions (calling functions across source files translated using C and C++ translators is more complex).

x.h
```
inline int my_abs(int p)
{
return (p < 0) ? -p : p;
}
```

x.c

```
#include "x.h"

extern inline int my_abs(int);
```

The handling of the second declaration of the function `my_abs` in `x.c` differs between C and C++. In C the presence of the **`extern`** storage-class specifier causes the definition to serve as a non-inline definition. While in C++ the presence of this storage-class specifier is redundant. The final result is to satisfy the requirement for exactly one non-inline definition in C, and to satisfy C++'s one definition rule. 1350 C++ one definition rule

---

1533 If a function is declared with an **`inline`** function specifier, then it shall also be defined in the same translation unit.

C++

> *An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case (3.2).* 7.1.2p4

The C++ Standard only requires the definition to be given if the function is used. A declaration of an inline function with no associated use does not require a definition. This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation.

1535 Inline substitution is not textual substitution, nor does it create a new function.

C++

The C++ Standard does not make this observation.

1536 Therefore, for example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appears, and not where the function is called;

C++

The C++ Standard does not make this observation.

1538 Likewise, the function has a single address, regardless of the number of inline definitions that occur in addition to the external definition.

C++

> *An **`inline`** function with external linkage shall have the same address in all translation units.* 7.1.2p4

There is no equivalent statement, in the C++ Standard, for inline functions having internal linkage.

1540 If all of the file scope declarations for a function in a translation unit include the **`inline`** function specifier without **`extern`**, then the definition in that translation unit is an *inline definition*. inline definition

C++

This term, or an equivalent one, is not defined in the C++ Standard.

The C++ Standard supports the appearance of more than one inline function definition, in a program, having a declaration with **`extern`**. This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation.

An inline definition provides an alternative to an external definition, which a translator may use to implement any call to the function in the same translation unit. 1542

C++

In C++ there are no alternatives, all inline functions are required to be the same.

7.1.2p4 *If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required.*

A C program may contain a function, with external linkage, that is declared inline in one translation unit but not be declared inline in another translation unit. When such a program is translated using a C++ translator a diagnostic may be issued.

It is unspecified whether a call to the function uses the inline definition or the external definition.[120)] 1543

C++

In the C++ Standard there are no alternatives. An inline definition is always available and has the same definition:

7.1.2p4 *An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case (3.2).*

Rationale Second, the requirement that all definitions of an inline function be "exactly the same" is replaced by the requirement that the behavior of the program should not depend on whether a call is implemented with a visible inline definition, or the external definition, of a function. This allows an inline definition to be specialized for its use within a particular translation unit. For example, the external definition of a library function might include some argument validation that is not needed for calls made from other functions in the same library. These extensions do offer some advantages; and programmers who are concerned about compatibility can simply abide by the stricter C++ rules.

EXAMPLE inline

EXAMPLE The declaration of an inline function with external linkage can result in either an external definition, or a definition available for use only within the translation unit. A file scope declaration with **extern** creates an external definition. The following example shows an entire translation unit. 1544

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

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

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

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

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

C++

The declaration:

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

does not create a reference to an external definition in C++.

1546 120) Since an inline definition is distinct from the corresponding external definition and from any other corresponding inline definitions in other translation units, all corresponding objects with static storage duration are also distinct in each of the definitions. footnote 120

C++

> *A **static** local variable in an extern inline function always refers to the same object. A string literal in an **extern inline** function is the same object in different translation units.* 7.1.2p4

The C++ Standard is silent about the case where the **extern** keyword does not appear in the declaration.

```
inline const char *saddr(void)
{
static const char name@lsquare[]@rsquare[] = "saddr";
return name;
}

int compare_name(void)
{
return saddr() == saddr(); /* may use extern definition in one case and inline in the other */
                           // They are either the same or the program is
                           // in violation of 7.1.2p2 (no diagnostic required)
}
```

6.7.5 Declarators

1547 declarator syntax

declarator:
    pointer$_{opt}$ *direct-declarator*
direct-declarator:
    identifier
    (*declarator* **)**
    direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
    direct-declarator **[static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
    direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**
    direct-declarator **[** *type-qualifier-list*$_{opt}$ * **]**
    direct-declarator **(** *parameter-type-list* **)**
    direct-declarator **(** *identifier-list*$_{opt}$ **)**
pointer:
    * *type-qualifier-list*$_{opt}$
    * *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier
parameter-type-list:
    parameter-list
    parameter-list **, ...**
parameter-list:
    parameter-declaration
    parameter-list **,** *parameter-declaration*
parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator$_{opt}$
identifier-list:
    identifier
    identifier-list **,** *identifier*

C90

Support for the syntax:

direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
direct-declarator **[static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list*$_{opt}$ * **]**

is new in C99. Also the C90 Standard only supported the form:

direct-declarator **[** *constant-expression*$_{opt}$ **]**

C++

The syntax:

direct-declarator **[** *type-qualifier-list*$_{opt}$ *assignment-expression*$_{opt}$ **]**
direct-declarator **[static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list* **static** *assignment-expression* **]**
direct-declarator **[** *type-qualifier-list*$_{opt}$ * **]**
direct-declarator **(** *identifier-list*$_{opt}$ **)**

is not supported in C++ (although the form *direct-declarator [constant-expression*$_{opt}$ *]* is supported).

The C++ Standard also supports (among other constructions) the form:

8p4 *direct-declarator (parameter-declaration-clause) cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$

The C++ Standard also permits the comma before an ellipsis to be omitted, e.g., `int f(int a ...);`.

Semantics

1549 A *full declarator* is a declarator that is not part of another declarator. full declarator

C90

Although this term was used in the C90 Standard, in translation limits, it was not explicitly defined.

C++

This term, or an equivalent one, is not defined by the C++ Standard.

1550 The end of a full declarator is a sequence point. full declarator sequence point

C90

The ability to use an expression causing side effects in an array declarator is new in C99. Without this construct there is no need to specify a sequence point at the end of a full declarator.

C++

The C++ Standard does not specify that the end of ant declarator is a sequence point. This does not appear to result in any difference of behavior.

1551 If in the nested sequence of declarators in a full declarator ~~contains~~ there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be *variably modified*. variably modified

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

Implementation limits

1558 As discussed in 5.2.4.1, an implementation may limit the number of pointer, array, and function declarators that modify an arithmetic, structure, union, or incomplete type, either directly or via one or more **typedef**s. declarator complexity limits

C90

> *The implementation shall allow the specification of types that have at least 12 pointer, array, and function declarators (in any valid combinations) modifying an arithmetic, a structure, a union, or an incomplete type, either directly or via one or more* ***typedef****s.*

6.7.5.1 Pointer declarators

Semantics

1560 If, in the declaration "**T D1**", **D1** has the form derived-declarator-type-list

    * type-qualifier-list_opt D

and the type specified for *ident* in the declaration "**T D**" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list type-qualifier-list* pointer to *T*".

C++

The C++ Standard uses the term *cv-qualifier-seq* instead of *type-qualifier-list*.

pointer types to be compatible

For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types. 1562

C++

The C++ Standard does not define the term *compatible type*, although in the case of qualified pointer types the term *similar* is defined (4.4p4). When two pointer types need to interact the C++ Standard usually specifies that the qualification conversions (clause 4) are applied and then requires that the types be the same. These C++ issues are discussed in the sentences in which they occur.

6.7.5.2 Array declarators

Constraints

In addition to optional type qualifiers and the keyword **static**, the [and] may delimit an expression or *. 1564

C90

> *The expression delimited by* **[** *and* **]** *(which specifies the size of an array) shall be an integral constant expression that has a value greater than zero.*

Support for the optional type qualifiers, the keyword **static**, the expression not having to be constant, and support for * between [and] in a declarator is new in C99.

C++

Support for the optional type qualifiers, the keyword **static**, the expression not having to be constant, and * between [and] in a declarator is new in C99 and is not specified in the C++ Standard.

array element not incomplete type
array element not function type

The element type shall not be an incomplete or function type. 1567

C90

array contiguously allocated set of objects

In C90 this wording did not appear within a Constraints clause. The requirement for the element type to be an object type appeared in the description of array types, which made violation of this requirement to be undefined behavior. The undefined behavior of all known implementations was to issue a diagnostic, so no actual difference in behavior between C90 and C99 is likely to occur.

C++

8.3.4p1

> *T is called the array element type; this type shall not be a reference type, the (possibly cv-qualified) type* **void***, a function type or an abstract class type.*

The C++ Standard does not disallow incomplete element types (apart from the type **void**). This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation.

array parameter qualifier only in outermost

The optional type qualifiers and the keyword **static** shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation. 1568

C90

Support for use of type qualifiers and the keyword **static** in this context is new in C99.

C++

Support for use of type qualifiers and the keyword **static** in this context is new in C99 is not supported in C++.

1569 ~~Only an~~ An ordinary identifier (as defined in 6.2.3) ~~with both block scope or function prototype scope and no linkage shall have a variably modified type.~~ that has a variably modified type shall have either block scope and no linkage or function prototype scope. variable modified only scope

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

Semantics

1571 If, in the declaration "**T D1**", **D1** has one of the forms: qualified array of

```
D[ type-qualifier-list_opt assignment-expression_opt ]
D[ static type-qualifier-list_opt assignment-expression ]
D[ type-qualifier-list static assignment-expression ]
D[ type-qualifier-list_opt * ]
```

and the type specified for *ident* in the declaration "**T D**" is "*derived-declarator-type-list T*", then the type specified for *ident* is "*derived-declarator-type-list* array of *T*".[121)]

C90

Support for forms other than:

```
D[ constant-expression_opt ]
```

is new in C99.

C++

The C++ Standard only supports the form:

```
D[ constant-expression_opt ]
```

A C++ translator will issue a diagnostic if it encounters anything other than a constant expression between the **[** and **]** tokens.

The type of the array is also slightly different in C++, which include the number of elements in the type:

> *. . . the array has N elements numbered 0 to N-1, and the type of the identifier of D is "derived-declarator-type-list array of N T."* 8.3.4p1

1573 If the size is not present, the array type is an incomplete type. array incomplete type

C++

The C++ Standard classifies all compound types as object types. It uses the term *incomplete object type* to refer to this kind of type. 475 object types

> *If the constant expression is omitted, the type of the identifier of D is "derived-declarator-type-list array of unknown bound of T," an incomplete object type.* 8.3.4p1

variable length array specified by *

If the size is * instead of being an expression, the array type is a variable length array type of unspecified size, which can only be used in declarations with function prototype scope;[122] 1574

C90

Support for a size specified using the * token is new in C99.

C++

Specifying a size using the * token is new in C99 and is not available in C++.

variable length array type

If the size is an integer constant expression and the element type has a known constant size, the array type is not a variable length array type; 1576

C90

Support for specifying a size that is not an integer constant expression is new in C99.

C++

Support for specifying a size that is not an integer constant expression is new in C99 and is not specified in the C++ Standard.

If the size is an expression that is not an integer constant expression: 1580

C90

Support for non integer constant expressions in this context is new in C99.

C++

Support for non integer constant expressions in this context is new in C99 and is not available in C++.

sizeof VLA unspecified evaluation

Where a size expression is part of the operand of a `sizeof` operator and changing the value of the size expression would not affect the result of the operator, it is unspecified whether or not the size expression is evaluated. 1584

C90

The operand of `sizeof` was not evaluated in C90. With the introduction of variable length arrays it is possible that the operand will need to be evaluated in C99.

array type to be compatible

For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present, and are integer constant expressions, then both size specifiers shall have the same constant value. 1585

C++

compatible type if 631

The C++ Standard does not define the concept of compatible type, it requires types to be the same.

If the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values. 1586

C90

The number of contexts in which array types can be incompatible has increased in C99, but the behavior is intended to be the same.

C++

There are no cases where the C++ Standard discusses a requirement that two arrays have the same number of elements.

6.7.5.3 Function declarators (including prototypes)

Constraints

1593 The only storage-class specifier that shall occur in a parameter declaration is **register**. parameter storage-class

C++

> *The **auto** or **register** specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4).* 7.1.1p2

C source code developed using a C++ translator may contain the storage-class specifier **auto** applied to a parameter. However, usage of this keyword is rare (see Table **??**) and in practice it is very unlikely to occur in this context.

The C++ Standard covers the other two cases with rather sloppy wording.

> *There can be no **static** function declarations within a block, nor any **static** function parameters.* 7.1.1p4

> *The **extern** specifier cannot be used in the declaration of class members or function parameters.* 7.1.1p5

1594 An identifier list in a function declarator that is not part of a definition of that function shall be empty.

C++

The C++ Standard does not support the old style of C function declarations.

1595 After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type. parameter adjustment in definition

C90

The C90 Standard did not explicitly specify that the check on the parameter type being incomplete occurred “after adjustment”.

C++

The C++ Standard allows a few exceptions to the general C requirement:

> *If the type of a parameter includes a type of the form “pointer to array of unknown bound of `T`” or “reference to array of unknown bound of `T`,” the program is ill-formed.[87)]* 8.3.5p6

> *This excludes parameters of type “`ptr-arr-seq T2`” where `T2` is “pointer to array of unknown bound of `T`” and where `ptr-arr-seq` means any sequence of “pointer to” and “array of” derived declarator types. This exclusion applies to the parameters of the function, and if a parameter is a pointer to function or pointer to member function then to its parameters also, etc.* Footnote 87

> *The parameter list `(void)` is equivalent to the empty parameter list. Except for this special case, **void** shall not be a parameter type (though types derived from **void**, such as `void*`, can).* 8.3.5p2

8.3.5p6

The type of a parameter or the return type for a function declaration that is not a definition may be an incomplete class type.

```
void f(struct s1_tag ** p1) /* incomplete type, constraint violation */
                            // defined behavior
{
struct s2_tag **loc; /* Size not needed, so permitted. */
}
```

Semantics

If, in the declaration “`T D1`”, `D1` has the form 1596

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

or

`D(` *identifier-list*$_{opt}$ `)`

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

C++

The form supported by the C++ Standard is:

8.3.5p1 ***D1* (** *parameter-declaration-clause* **)** *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$

The term used for the identifier in the C++ Standard is:

8.3.5p1 *“derived-declarator-type-list function of (parameter-declaration-clause) cv-qualifier-seq*^{opt} *returning T”;*

The old-style function definition *D(* *identifier-list*$_{opt}$ *)* is not supported in C++.

8.3.5p2 *If the parameter-declaration-list is empty, the function takes no arguments. The parameter list* `(void)` *is equivalent to the empty parameter list.*

The C syntax treats `D()` as an instance of an empty identifier-list, while the C++ syntax treats it as an empty *parameter-type-list*. Using a C++ translator to translate source containing this form of function declaration may result a diagnostic being generated when the declared function is called (if it specifies any arguments).

A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function. 1597

C++

8.3.5p2 *The* `parameter-declaration-clause` *determines the arguments that can be specified, and their processing, when the function is called.*

1598 A declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*", where the type qualifiers (if any) are those specified within the [and] of the array type derivation. array type adjust to pointer to

C90

Support for type qualifiers between [and], and the consequences of their use, is new in C99.

C++

This adjustment is performed in C++ (8.3.5p3) but the standard does not support the appearance of type qualifiers between [and].

Source containing type qualifiers between [and] will cause a C++ translator to generate a diagnostic.

1599 If the keyword **static** also appears within the [and] of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression. function declarator static

C90

Support for the keyword **static** in this context is new in C99.

C++

Support for the keyword **static** in this context is new in C99 and is not available in C++.

1601 If the list terminates with an ellipsis (, ...), no information about the number or types of the parameters after the comma is supplied.[123)] ellipsis supplies no information

C++

The C++ Standard does not make this observation.

1602 The special case of an unnamed parameter of type **void** as the only item in the list specifies that the function has no parameters. parameter type void

C90

The C90 Standard was reworded to clarify the intent by the response to DR #157.

1603 If, iIn a parameter declaration, ~~a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator.~~ an identifier can be treated as a typedef name or as a parameter name, it shall be taken as a typedef name. parameter declaration typedef name in parentheses

C90

The response to DR #009 proposed adding the requirement: "If, in a parameter declaration, an identifier can be treated as a typedef name or as a parameter name, it shall be taken as a typedef name."

1604 If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the [*] notation in their sequences of declarator specifiers to specify variable length array types.

C90

Support for the [*] notation is new in C99.

C++

The wording:

> *If the type of a parameter includes a type of the form "pointer to array of unknown bound of T" or "reference to array of unknown bound of T," the program is ill-formed.*[87)] 8.3.5p6

does not contain an exception for the case of a function declaration.
Support for the `[*]` notation is new in C99 and is not specified in the C++ Standard.

An identifier list declares only the identifiers of the parameters of the function. 1606

C++

This form of function declarator is not available in C++.

---

function declarator empty list

The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.[124] 1608

C++

The following applies to both declarations and definitions of functions:

8.3.5p2 *If the `parameter-declaration-clause` is empty, the function takes no arguments.*

A call made within the scope of a function declaration that specifies an empty parameter list, that contains arguments will cause a C++ translator to issue a diagnostic.

---

function compatible types

For two function types to be compatible, both shall specify compatible return types.[125] 1611

C++

compatible type if 631

The C++ Standard does not define the concept of compatible type, it requires types to be the same.

3.5p10 *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, . . .*

8.3.5p3 *All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the function type.*

If one return type is an enumerated type and the another return type is the compatible integer type. C would consider the functions compatible. C++ would not consider the types as agreeing exactly.

Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; 1612

C++

A parameter type list is always present in C++, although it may be empty.

---

corresponding parameters shall have compatible types. 1613

C++

8.3.5p3 *All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the function type.*

compatible type if 631

The C++ Standard does not define the concept of compatible type, it requires types to be the same. If one parameter type is an enumerated type and the corresponding parameter type is the corresponding compatible integer type. C would consider the functions to be compatible, but C++ would not consider the types as being the same.

1614 If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions.

C++

The C++ Standard does not support the C identifier list form of parameters. An empty parameter list is interpreted differently:

> *If the `parameter-declaration-clause` is empty, the function takes no arguments.* 8.3.5p2

The two function declarations do not then agree:

> *After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, . . .* 3.5p10

A C++ translator is likely to issue a diagnostic if two declarations of the same function do not agree (the object code file is likely to contain function signatures, which are based on the number and type of the parameters in the declarations).

1615 If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier.

C++

The C++ Standard does not support the C identifier list form of parameters.

> *If the `parameter-declaration-clause` is empty, the function takes no arguments.* 8.3.5p2

The C++ Standard requires that a function declaration always be visible at the point of call (5.2.2p2). Issues involving argument promotion do not occur (at least for constructs supported in C).

```
void  f(int, char);
void  f(char, int);
char a, b;
f(a,b);             // illegal: Which function is called?  Both fit
                    // equally well (equally badly).
```

1616 (In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.) parameter qualifier in composite type

C90

The C90 wording:

> *(For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type, as in 6.7.1. For each parameter declared with qualified type, its type for these comparisons is the unqualified version of its declared type.)*

was changed by the response to DR #013 question 1 (also see DR #017q15 and DR #040q1).

C++

compos-ite type 642 compati-ble type 631 if

The C++ Standard does not define the term *composite type*. Neither does it define the concept of compatible type, it requires types to be the same.

The C++ Standard transforms the parameters' types and then:

8.3.5p3

> *If a* `storage-class-specifier` *modifies a parameter type, the specifier is deleted. [Example:* `register char*` *becomes* `char*` *—end example] Such* `storage-class-specifiers` *affect only the definition of the parameter within the body of the function; they do not affect the function type. The resulting list of transformed parameter types is the function's parameter type list.*

It is this parameter type list that is used to check whether two declarations are the same.

EXAMPLE function returning pointer to

EXAMPLE 1 The declaration 1617

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

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

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

C++

Function declared with an empty parameter type list are considered to take no arguments in C++.

---

footnote 125

125) If both function types are "old style", parameter types are not compared. 1620

C++

The C++ Standard does not support *old style* function types.

EXAMPLE 4 The following prototype has a variably modified parameter. 1621

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

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

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

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

6.7.6 Type names

abstract declarator syntax

1624
type-name:
    specifier-qualifier-list abstract-declarator$_{opt}$
abstract-declarator:
    pointer
    pointer$_{opt}$ *direct-abstract-declarator*
direct-abstract-declarator:
    (*abstract-declarator* **)**
    ~~*direct-abstract-declarator*$_{opt}$ **[** *assignment-expression* $_{opt}$ **]**~~
    direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list*$_{opt}$ *assignment-expression* $_{opt}$ **]**
    direct-abstract-declarator$_{opt}$ **[static** *type-qualifier-list*$_{opt}$ *assignment-expression* **]**
    direct-abstract-declarator$_{opt}$ **[** *type-qualifier-list* **static** *assignment-expression* **]**
    direct-abstract-declarator$_{opt}$ **[*]**
    direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

C90

Support for the form:

direct-abstract-declarator$_{opt}$ **[*]**

is new in C99. In the form:

direct-abstract-declarator$_{opt}$ **[** *assignment-expression*^{opt} **]**

C90 only permitted *constant-expression*$_{opt}$ to appear between **[** and **]**.

C++

The C++ Standard supports the C90 forms. It also includes the additional form (8.1p1):

direct-abstract-declarator$_{opt}$ **(** *parameter-declaration-clause* **)**
cv-qualifier-seq$_{opt}$ *exception-specification*$_{opt}$

Semantics

1626 This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the identifier.[126)]

C++

Restrictions on the use of type names in C++ are discussed elsewhere.

1118 sizeof constraints
1134 cast scalar or void type
function declarator return type

EXAMPLE abstract declarators

1627 EXAMPLE The constructions

```
(a)     int
(b)     int *
(c)     int *[3]
(d)     int (*)[3]
(e)     int (*)[*]
(f)     int *()
(g)     int (*)(void)
(h)     int (*const [])(unsigned int, ...)
```

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

C90

Support for variably length arrays is new in C99.

C++

Support for variably length arrays is new in C99 and is not specified in the C++ Standard.

6.7.7 Type definitions

Constraints

If a typedef name specifies a variably modified type then it shall have block scope. 1630

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and not specified in the C++ Standard.

Semantics

---

array size evaluated when declaration reached

Any array size expressions associated with variable length array declarators are evaluated each time the declaration of the typedef name is reached in the order of execution. 1632

C90

Support for variable length array declarators is new in C99.

C++

Support for variable length array declarators is new in C99 and is not specified in the C++ Standard.

That is, in the following declarations: 1634

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

type_ident is defined as a typedef name with the type specified by the declaration specifiers in **T** (known as *T*), and the identifier in **D** has the type "*derived-declarator-type-list T*" where the *derived-declarator-type-list* is specified by the declarators of **D**.

C++

This example and its associated definition of terms is not given in the C++ Standard.

6.7.8 Initialization

initialization syntax

1641

```
initializer:
        assignment-expression
        { initializer-list }
        { initializer-list , }
initializer-list:
        designation_opt initializer
        initializer-list , designation_opt initializer
designation:
```

```
                designator-list =
designator-list:
                designator

                designator-list designator
designator:
                [ constant-expression ]
                . identifier
```

C90

Support for designators in initializers is new in C99.

C++

Support for designators in initializers is new in C99 and is not specified in the C++ Standard.

Constraints

1642 No initializer shall attempt to provide a value for an object not contained within the entity being initialized. initializer value not contained in object

C90

> *There shall be no more initializers in an initializer list than there are objects to be initialized.*

Support for designators in initializers is new in C99 and a generalization of the wording is necessary to cover the case of a name being used that is not a member of the structure or union type, or an array index that does not lie within the bounds of the object array type.

C++

The C++ Standard wording has the same form as C90, because it does not support designators in initializers.

> *An* `initializer-list` *is ill-formed if the number of initializers exceeds the number of members or elements to initialize.* 8.5.1p6

1643 The type of the entity to be initialized shall be an array of unknown size or an object type that is not a variable length array type.

C90

Support for variable length array types is new in C99.

1644 All the expressions in an initializer for an object that has static storage duration shall be constant expressions or string literals. initializer static storage duration object

C90

> *All the expressions in an initializer for an object that has static storage duration or in an initializer list for an object that has aggregate or union type shall be constant expressions.*

C99 has relaxed the requirement that aggregate or union types always be initialized with constant expressions.

Support for string literals in this context was added by the response to DR #150.

C++

8.5p2
> *Automatic, register, static, and external variables of namespace scope can be initialized by arbitrary expressions involving literals and previously declared variables and functions.*

A program, written using only C constructs, could be acceptable to a conforming C++ implementation, but not be acceptable to a C implementation.

C++ translators that have an *operate in C mode* option have been known to fail to issue a diagnostic for initializers that would not be acceptable to conforming C translators.

---

identifier linkage at block scope

If the declaration of an identifier has block scope, and the identifier has external or internal linkage, the declaration shall have no initializer for the identifier. 1645

C++

The C++ Standard does not specify any equivalent constraint.

---

designator constant-expression

If a designator has the form 1646

```
[ constant-expression ]
```

then the current object (defined below) shall have array type and the expression shall be an integer constant expression.

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and is not specified in the C++ Standard.

---

designator . identifier

If a designator has the form 1648

```
. identifier
```

then the current object (defined below) shall have structure or union type and the identifier shall be the name of a member of that type.

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and is not specified in the C++ Standard.

Semantics

---

unnamed members initialization of

Except where explicitly stated otherwise, for the purposes of this subclause unnamed members of objects of structure and union type do not participate in initialization. 1650

C90

The C90 Standard wording "All unnamed structure or union members are ignored during initialization." was modified by the response to DR #017q17.

C++

This requirement is not explicitly specified in the C++ Standard.

1651 Unnamed members of structure objects have indeterminate value even after initialization.

C90

This was behavior was not explicitly specified in the C90 Standard.

C++

This behavior is not explicitly specified in the C++ Standard.

1653 If an object that has static storage duration is not initialized explicitly, then: static initialization default value

C++

The C++ Standard specifies a two-stage initialization model. The final result is the same as that specified for C.

> *The memory occupied by any object of static storage duration shall be zero-initialized at program startup before any other initialization takes place. [Note: in some cases, additional initialization is done later.]* 8.5p6

1655 — if it has arithmetic type, it is initialized to (positive or unsigned) zero;

C90

The distinction between the signedness of zero is not mentioned in the C90 Standard.

1657 — if it is a union, the first named member is initialized (recursively) according to these rules.

C90

This case was not called out in the C90 Standard, but was added by the response in DR #016.

C++

The C++ Standard, 8.5p5, specifies the first data member, not the first named data member.

1658 The initializer for a scalar shall be a single expression, optionally enclosed in braces. initializer scalar

C++

> *If T is a scalar type, then a declaration of the form* 8.5p13
>
> ```
> T x = { a };
> ```
>
> *is equivalent to*
>
> ```
> T x = a;
> ```

This C++ specification is not the same the one in C, as can be seen in:

```
struct DR_155 {
                 int i;
               } s = { { 1 } }; /* does not affect the conformance status of the program */
                                // ill-formed
```

8.5p14 *If the conversion cannot be done, the initialization is ill-formed.*

While a C++ translator is required to issue a diagnostic for a use of this ill-formed construct, such an occurrence causes undefined behavior in C (the behavior of many C translators is to issue a diagnostic).

The initial value of the object is that of the expression (after conversion); 1659

C90

The C90 wording did not include "(after conversion)". Although all known translators treated initializers just like assignment and performed the conversion.

---

initializer type constraints

the same type constraints and conversions as for simple assignment apply, taking the type of the scalar to be the unqualified version of its declared type. 1660

C++

Initialization is not the same as simple assignment in C++ (5.17p5, 8.5p14). However, if only the constructs available in C are used the behavior is the same.

The initializer for a structure or union object that has automatic storage duration shall be either an initializer list as described below, or a single expression that has compatible structure or union type. 1662

C++

The C++ Standard permits an arbitrary expression to be used in all contexts (8.5p2).

This difference permits a program, written using only C constructs, to be acceptable to a conforming C++ implementation but not be acceptable to a C implementation. C++ translators that have an *operate in C mode* switch do not always diagnose initializers that would not be acceptable to all conforming C translators.

---

initializing including unnamed members

In the latter case, the initial value of the object, including unnamed members, is that of the expression. 1663

C++

The C++ Standard does not contain this specification.

---

initialize uses successive characters from string literal

Successive characters of the character string literal (including the terminating null character if there is room or if the array is of unknown size) initialize the elements of the array. 1665

C++

The C++ Standard does not specify that the terminating null character is optional, as is shown by an explicit example (8.5.2p2).

An object initialized with a string literal whose terminating null character is not included in the value used to initialize the object, will cause a diagnostic to be issued by a C++ translator.

```
char hello@lsquare[]5@rsquare[] = "world"; /* strictly conforming */
                                // ill-formed
```

---

current object brace enclosed initializer

Each brace-enclosed initializer list has an associated *current object*. 1669

C90

The concept of *current object* is new in C99.

C++

The concept of *current object* is new in C99 and is not specified in the C++ Standard. It is not needed because the ordering of the initializer is specified (and the complications of designation initializers don't exist, because they are not supported):

> *. . . , written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the subaggregate.* 8.5.1p2

1670 When no designations are present, subobjects of the current object are initialized in order according to the type of the current object: array elements in increasing subscript order, structure members in declaration order, and the first named member of a union.[127)] initialization no designator member ordering

C90

> *Otherwise, the initializer for an object that has aggregate type shall be a brace-enclosed list of initializers for the members of the aggregate, written in increasing subscript or member order; and the initializer for an object that has union type shall be a brace-enclosed initializer for the first member of the union.*

The wording specifying named member was added by the response to DR #016.

1671 In contrast, a designation causes the following initializer to begin initialization of the subobject described by the designator. initialization using a declarator

C90

Support for designations is new in C99.

C++

Support for designations is new in C99, and is not available in C++

1673 Each designator list begins its description with the current object associated with the closest surrounding brace pair. designator list current object

C90

Support for designators is new in C99.

C++

Support for designators is new in C99, and is not available in C++

1676 The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject;[130)] initialization in list order

C++

The C++ Standard does not support designators and so it is not possible to specify more than one initializer for an object.

1677 all subobjects that are not initialized explicitly shall be initialized implicitly the same as objects that have static storage duration. object initialized but not explicitly

C++

The C++ Standard specifies that objects having static storage duration are zero-initialized (8.5p6), while members that are not explicitly initialized are default-initialized (8.5.1p7). If constructs that are only available in C are used the behavior is the same as that specified in the C Standard.

---

any remaining initializers are left to initialize the next element or member of the aggregate of which the current subaggregate or contained union is a part. 1681

C90

This behavior was not pointed out in the C90 Standard.

---

initializer fewer in list than members

If there are fewer initializers in a brace-enclosed list than there are elements or members of an aggregate, or fewer characters in a string literal used to initialize an array of known size than there are elements in the array, the remainder of the aggregate shall be initialized implicitly the same as objects that have static storage duration. 1682

C90

The string literal case was not explicitly specified in the C90 Standard, but was added by the response to DR #060.

C++

The C++ Standard specifies that objects having static storage duration are zero-initialized (8.5p6), while members that are not explicitly initialized are default-initialized (8.5.1p7). If only constructs available in C are used the behavior is the same as that specified in the C Standard.

---

array of unknown size initialized

If an array of unknown size is initialized, its size is determined by the largest indexed element with an explicit initializer. 1683

C90

> *If an array of unknown size is initialized, its size is determined by the number of initializers provided for its elements.*

Support for designators is new in C99.

---

initializer completes incomplete array type

At the end of its initializer list, the array no longer has incomplete type. 1684

C++

The C++ Standard does not specify how an incomplete array type can be completed. But the example in 8.5.1p4 suggests that with an object definition, using an incomplete array type, the initializer creates a new array type. The C++ Standard seems to create a new type, rather than completing the existing incomplete one (which is defined, 8.3.4p1, as being a different type).

---

footnote 127

127) If the initializer list for a subaggregate or contained union does not begin with a left brace, its subobjects are initialized as usual, but the subaggregate or contained union does not become the current object: current objects are associated only with brace-enclosed initializer lists. 1685

C90

The concept of current object is new in C99.

C++

The concept of current object is new in C99 and is not specified in the C++ Standard.

1686 128) After a union member is initialized, the next object is not the next member of the union; footnote 128

C90

The C90 Standard explicitly specifies, in all the relevant places in the text, that only the first member of a union is initialized.

1691 The order in which any side effects occur among the initialization list expressions is unspecified.[131)]

C90

The C90 Standard requires that the expressions used to initialize an aggregate or union be constant expressions. Whatever the order of evaluation used the external behavior is likely to be the same (it is possible that one or more members of a structure type are volatile-qualified).

C++

The C++ Standard does not explicitly specify any behavior for the order of side effects among the initialization list expressions (which implies unspecified behavior in this case).

1692 EXAMPLE 1 Provided that **<complex.h>** has been **#include**d, the declarations

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

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

C90

Support for complex types is new in C99.

C++

The C++ Standard does not define an identifier named **I** in **<complex>**.

1697 131) In particular, the evaluation order need not be the same as the order of subobject initialization. footnote 131

C++

The C++ Standard does explicitly specify the ordering of side effects among the expressions contained in an initialization list.

1699 EXAMPLE 7
One form of initialization that completes array types involves typedef names. Given the declaration

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

the declaration

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

is identical to

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

due to the rules for incomplete types.

C++

The C++ Standard does not explicitly specify this behavior.

EXAMPLE array initialization

EXAMPLE 8 The declaration 1700

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

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

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

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

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

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

C++

The initializer used in the declaration of `t` would cause a C++ translator to issue a diagnostic. It is not equivalent to the alternative, C, form given below it.

EXAMPLE 9 Arrays can be initialized to correspond to the elements of an enumeration by using designators: 1701

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

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

EXAMPLE div_t

EXAMPLE 10 Structure members can be initialized to nonzero values without depending on their order: 1702

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

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

EXAMPLE designators with inconsistently brackets

EXAMPLE 11 Designators can be used to provide explicit initialization when unadorned initializer lists might be misunderstood: 1703

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

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

1704 EXAMPLE 12 Space can be "allocated" from both ends of an array by using a single designator: EXAMPLE overriding values

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

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

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

1705 EXAMPLE 13 Any member of a union can be initialized:

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

C90

Support for designators is new in C99.

C++

Support for designators is new in C99 and they are not specified in the C++ Standard.

6.8 Statements and blocks

1707 statement syntax

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

C++

In C++ local declarations are classified as statements (6p1). They are called *declaration statements* and their syntax nonterminal is *declaration-statement*.

Semantics

1708 A *statement* specifies an action to be performed. statement

C++

The C++ Standard does not make this observation.

1710 A *block* allows a set of declarations and statements to be grouped into one syntactic unit. block

C90

> *A compound statement (also called a block) allows a set of statements to be grouped into one syntactic unit, which may have its own set of declarations and initializations (as discussed in 6.1.2.4).*

The term *block* includes compound statements and other constructs, that need not be enclosed in braces, in C99. The differences associated with these constructs is called out in the sentences in which they occur.

1729 compound statement syntax

C++

The C++ Standard does not make this observation.

---

object initializer evaluated when

The initializers of objects that have automatic storage duration, and the variable length array declarators of ordinary identifiers with block scope, are evaluated and the values are stored in the objects (including storing an indeterminate value in objects without an initializer) each time the declaration is reached in the order of execution, as if it were a statement, and within each declaration in the order that declarators appear. 1711

C90

Support for variable length array types is new in C99.

C++

Support for variable length array types is new in C99 and they are not specified in the C++ Standard.

---

full expression

A *full expression* is an expression that is not part of another expression or of a declarator. 1712

C++

1.9p12 *A full-expression is an expression that is not a subexpression of another expression.*

The C++ Standard does not include declarators in the definition of a full expression. Source developed using a C++ translator may contain declarations whose behavior is undefined in C.

```
int i;

void f(void)
{
int a@lsquare[]i++@rsquare[]@lsquare[]i++@rsquare[]; /* undefined behavior */
                 // a sequence point between modifications to i
}
```

Each of the following is a full expression: 1713

C++

The C++ Standard does not enumerate the constructs that are full expressions.

---

full expression expression statement

the expression in an expression statement; 1715

C++

The following is not the same as saying that an expression statement is a full expression, but it shows the effect is the same:

6.2p1 *All side effects from an expression statement are completed before the next statement is executed.*

6.8.1 Labeled statements

Constraints

---

label name unique

Label names shall be unique within a function. 1725

C90

This wording occurred in Clause 6.1.2.1 Scope of identifiers in the C90 Standard. As such a program that contained non unique labels exhibited undefined behavior.

A function definition containing a non-unique label name that was accepted by some C90 translator (a very rare situation) will cause a C99 translator to generate a diagnostic.

Semantics

1727 Labels in themselves do not alter the flow of control, which continues unimpeded across them. case fall through

C++

The C++ Standard only explicitly states this behavior for **`case`** and **`default`** labels (6.4.2p6). It does not specify any alternative semantics for 'ordinary' labels.

6.8.2 Compound statement

1729 compound statement syntax

```
compound-statement:
                { block-item-list_opt }
block-item-list:
                block-item
                block-item-list block-item
block-item:
                declaration
                statement
```

C90

The C90 syntax required that declarations occur before statements and the two not be intermixed.

```
compound-statement:
                 { declaration-list_opt statement-list_opt }
declaration-list:
                declaration
                declaration-list declaration
statement-list:
                statement
                statement-list statement
```

C++

The C++ Standard uses the C90 syntax. However, the interpretation is the same as the C99 syntax because C++ classifies a declaration as a statement.

Semantics

1730 A *compound statement* is a block. compound statement is a block

C90

The terms *compound statement* and *block* were interchangeable in the C90 Standard.

6.8.3 Expression and null statements

Semantics

1735 132) Such as assignments, and function calls which have side effects. footnote 132

C++

The C++ Standard does not contain this observation.

6.8.4 Selection statements

Semantics

controlling expression if statement

A selection statement selects among a set of statements depending on the value of a controlling expression. 1740

C++

The C++ Standard omits to specify how the flows of control are selected:

6.4p1 *Selection statements choose one of several flows of control.*

block selection statement

A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. 1741

C90

See Commentary.

C++

The C++ behavior is the same as C90. See Commentary.

block selection substatement

Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement. 1742

C90

The following example illustrates the rather unusual combination of circumstances needed for the specification change, introduced in C99, to result in a change of behavior.

```
extern void f(int);
enum {a, b} glob;

int different(void)
{
if (glob == a)
                      /* No braces.                                          */
   f((enum {b, a})1); /* Declare identifiers with same name and compatible type. */

return b;  /* C90: numeric value 1 */
           /* C99: numeric value 0 */
}
```

C++

The C++ behavior is the same as C90.

6.8.4.1 The `if` statement

Constraints

if statement controlling expression scalar type

The controlling expression of an `if` statement shall have scalar type. 1743

C++

6.4p4

> *The value of a condition that is an expression is the value of the expression, implicitly converted to* **bool** *for statements other than* **switch***; if that conversion is ill-formed, the program is ill-formed.*

If only constructs that are available in C are used the set of possible expressions is the same.

Semantics

1744 In both forms, the first substatement is executed if the expression compares unequal to 0. if statement operand compare against 0

C++

The C++ Standard expresses this behavior in terms of true and false (6.4.1p1). The effect is the same.

1746 If the first substatement is reached via a label, the second substatement is not executed.

C++

The C++ Standard does not explicitly specify the behavior for this case.

6.8.4.2 The `switch` statement

Constraints

1748 The controlling expression of a **switch** statement shall have integer type. switch statement

C++

> *The condition shall be of integral type, enumeration type, or of a class type for which a single conversion function to integral or enumeration type exists (12.3).* 6.4.2p2

If only constructs that are available in C are used the set of possible expressions is the same.

1749 If a **switch** statement has an associated **case** or **default** label within the scope of an identifier with a variably modified type, the entire **switch** statement shall be within the scope of that identifier.[133] switch past variably modified type

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

The C++ Standard contains the additional requirement that (the wording in a subsequent example suggests that being visible rather than *in scope* is more accurate terminology):

> *It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps*[77)] *from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has POD type (3.9) and is declared without an* `initializer` *(8.5).* 6.7p3

```
void f(void)
{
switch (2)
   {
   int loc = 99; /* strictly conforming */
                 // ill-formed

   case 2: return;
   }
}
```

Semantics

Implementation limits

6.8.5 Iteration statements

iteration statement syntax

1763

iteration-statement:
while (*expression* **)** *statement*
do *statement* **while (** *expression* **) ;**
for ($expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*
for (*declaration* $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

C90

Support for the form:

for (*declaration* $expr_{opt}$ **;** $expr_{opt}$ **)** *statement*

is new in C99.

C++

The C++ Standard allows local variable declarations to appear within all conditional expressions. These can occur in **if**, **while**, and **switch** statements.

Constraints

for statement declaration part

The declaration part of a **for** statement shall only declare identifiers for objects having storage class **auto** or **register**. 1765

C90

Support for this functionality is new in C99.

C++

6.4p2 *The* `declarator` *shall not specify a function or an array. The* `type-specifier-seq` *shall not contain* ***typedef*** *and shall not declare a new class or enumeration.*

```
void f(void)
{
for (int la@lsquare[]10@rsquare[]; /* does not change the conformance status of the program */
                  // ill-formed
     ; ;)
   ;
for (enum {E1, E2} le; /* does not change the conformance status of the program */
                        // ill-formed
     ; ;)
   ;
for (static int ls; /* constraint violation */
                     // does not change the conformance status of the program
     ; ;)
   ;
}
```

Semantics

1766 An iteration statement causes a statement called the *loop body* to be executed repeatedly until the controlling expression compares equal to 0. iteration statement executed repeatedly loop body

C++

The C++ Standard converts the controlling expression to type **bool** and expresses the termination condition in terms of true and false. The final effect is the same as in C.

6.8.5.1 The while statement

6.8.5.2 The do statement

6.8.5.3 The for statement

1774 The statement for statement

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

behaves as follows:

C90

Except for the behavior of a **continue** *statement in the loop body, the statement*

```
for ( expression-1 ; expression-2 ; expression-3 ) statement
```

and the sequence of statements

```
expression-1 ;
while (expression-2) {
statement ;
expression-3 ;
}
```

are equivalent.

C++

Like the C90 Standard, the C++ Standard specifies the semantics in terms of an equivalent **while** statement. However, the C++ Standard uses more exact wording, avoiding the possible ambiguities present in the C90 wording.

1777 If *clause-1* is a declaration, the scope of any ~~variables~~ identifiers it declares is the remainder of the declaration and the entire loop, including the other two expressions;

C90

Support for this functionality is new in C99.

1780 Both *clause-1* and *expression-3* can be omitted.

C++

The C++ Standard does not make this observation, that can be deduced from the syntax.

6.8.6 Jump statements

Semantics

---

footnote 134

134) Thus, *clause-1* specifies initialization for the loop, possibly declaring one or more variables for use in the loop; 1784

C90

Support for declaring variables in this context is new in C99.

C++

The C++ Standard does not make this observation.

6.8.6.1 The goto statement

Constraints

---

goto past variably modified type

A **goto** statement shall not jump from outside the scope of an identifier having a variably modified type to inside the scope of that identifier. 1788

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and they are not specified in the C++ Standard. However, the C++ Standard contains the additional requirement that (the wording in a subsequent example suggests that being visible rather than *in scope* more accurately reflects the intent):

6.7p3 *A program that jumps[77)] from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has POD type (3.9) and is declared without an* `initializer` *(8.5).*

A C function that performs a jump into a block that declares an object with an initializer will cause a C++ translator to issue a diagnostic.

```
void f(void)
{
goto lab;    /* strictly conforming */
             // ill-formed
int loc = 1;

lab: ;
}
```

Semantics

6.8.6.2 The continue statement

Constraints

Semantics

More precisely, in each of the statements 1795

```
while (/* ... */) {    do {                      for (/* ... */) {
   /* ... */               /* ... */                  /* ... */
   continue;               continue;                  continue;
   /* ... */               /* ... */                  /* ... */
contin: ;               contin: ;                  contin: ;
}                      } while (/* ... */);      }
```

unless the **continue** statement shown is in an enclosed iteration statement (in which case it is interpreted within that statement), it is equivalent to **goto contin;**.[135)]

C++

The C++ Standard uses the example (6.6.2p1):

6.6.2p1

```
while (foo) {        do {              for (;;) {
  {                   {                 {
   // ...              // ...            // ...
  }                   }                 }
contin: ;            contin: ;         contin: ;
}                    } while (foo);    }
```

The additional brace-pair are needed to ensure that any necessary destructors (a construct not supported by C) are invoked.

6.8.6.3 The break statement

Constraints

Semantics

6.8.6.4 The `return` statement

Constraints

1799 A **return** statement with an expression shall not appear in a function whose return type is **void**. return void type

C++

6.6.3p3

*A return statement with an expression of type "cv **void**" can be used only in functions with a return type of cv void; the expression is evaluated just before the function returns to its caller.*

Source developed using a C++ translator may contain **return** statements with an expression returning a void type, which will cause a constraint violation if processed by a C translator.

```
void f(void)
{
return (void)4; /* constraint violation */
                // does not change the conformance status of a program
}
```

1800 A **return** statement without an expression shall only appear in a function whose return type is **void**. return without expression

C90

This constraint is new in C99.

```
int f(void)
{
return; /* Not a constraint violation in C90. */
}
```

Semantics

1802 A function may have any number of **return** statements.

C++

The C++ Standard does not explicitly specify this permission.

If a **return** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. 1803

C++

return void type 1799 The C++ Standard supports the use of expressions that do not return a value to the caller.

---

return implicit cast

If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.[136] 1804

C++

In the case of functions having a return type of *cv* **void** (6.6.3p3) the expression is not implicitly converted to that type. An explicit conversion is required.

---

footnote 136

136) The **return** statement is not an assignment. 1806

C90

This footnote did not appear in the C90 Standard. It was added by the response to DR #001.

C++

This distinction also occurs in C++, but as a special case of a much larger issue involving the creation of temporary objects (for constructs not available in C).

6.6.3p2 *A return statement can involve the construction and copy of a temporary object (12.2).*

12.2p1 *Temporaries of class type are created in various contexts: binding an rvalue to a reference (8.5.3), returning an rvalue (6.6.3), . . .*

6.9 External definitions

translation unit syntax external declaration syntax

1810

```
translation-unit:
                external-declaration
                translation-unit external-declaration
external-declaration:
                function-definition
                declaration
```

C++

The C++ syntax includes function definitions as part of declarations (3.5p1):

3.5p1 *translation-unit:* *declaration-seq*$_{opt}$

While the C++ Standard differs from C in supporting an empty translation unit, this is not considered a significant difference.

Constraints

1811 The storage-class specifiers **auto** and **register** shall not appear in the declaration specifiers in an external declaration. external declaration not auto/register

C++

The C++ Standard specifies where these storage-class specifiers can be applied, not where they cannot:

> *The* **auto** *or* **register** *specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4).* 7.1.1p2

A C++ translator is not required to issue a diagnostic if the storage-class specifiers **auto** and **register** appear in other scopes.

1813 Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a **sizeof** operator whose result is an integer constant), there shall be exactly one external definition for the identifier in the translation unit. internal linkage exactly one external definition

C++

The C++ Standard does not specify any particular linkage:

> *Every program shall contain exactly one definition of every non-inline function or object that is used in that program; no diagnostic required.* 3.2p3

The definition of the term used (3.2p2) also excludes operands of the **sizeof** operator.

Semantics

1814 As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations.

C++

The C++ Standard does not make this observation.

1815 These are described as "external" because they appear outside any function (and hence have file scope).

C++

The C++ Standard does not refer to them as "external" in the syntax.

1817 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object. external definition

C90

Support for inline definitions new in C99.

C++

The C++ Standard does not define the term *external definition*, or one equivalent to it.

1818 If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **sizeof** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier; external linkage exactly one external definition

C90

Support for the **sizeof** operator having a result that is not a constant expression is new in C99.

internal linkage 1813
exactly one external definition

C++

The specification given in the C++ is discussed elsewhere.

---

otherwise, there shall be no more than one.[137] 1819

C++

The C++ Standard does not permit more than one definition in any translation unit (3.2p1). However, if a non-inline function or an object is not used in a program it does not prohibit more than one definition in the set of translation units making up that program.

Source developed using a C++ translator may contain multiple definitions of objects that are not referred.

---

footnote 137

137) Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it. 1820

C++

The C++ Standard does not make this observation.

6.9.1 Function definitions

function definition syntax

1821

function-definition:
    declaration-specifiers declarator declaration-list$_{opt}$ *compound-statement*
declaration-list:
    declaration
    declaration-list declaration

C++

The C++ Standard does not support the appearance of *declarator-list*$_{opt}$. Function declarations must always use prototypes. It also specifies additional syntax for *function-definition*. This syntax involves constructs that are not available in C.

Constraints

The identifier declared in a function definition (which is the name of the function) shall have a function type, as specified by the declarator portion of the function definition.[138] 1822

C++

The C++ Standard specifies the syntax (which avoids the need for a footnote like that given in the C Standard):

8.4p1 *The* `declarator` *in a* `function-definition` *shall have the form*

> **D1 (** *parameter-declaration-clause* **)** *cv-qualifier-seq*$_{opt}$ *exception-specification*$_{opt}$

---

function definition return type

The return type of a function shall be **void** or an object type other than array type. 1823

C++

8.3.5p6 *Types shall not be defined in return or parameter types.*

The following example would cause a C++ translator to issue a diagnostic.

```
enum E {E1, E2} f (void) /* does not change the conformance status of program */
                         // ill-formed
{
return E1;
}
```

1824 The storage-class specifier, if any, in the declaration specifiers shall be either **extern** or **static**.

C++

*The **auto** or **register** specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4).* 7.1.1p2

A C++ translator is not required to issue a diagnostic if these storage-class specifiers appear in other contexts. Source developed using a C++ translator may contain constraint violations if processed by a C translator.

1825 If the declarator includes a parameter type list, the declaration of each parameter shall include an identifier, except for the special case of a parameter list consisting of a single parameter of type **void**, in which case there shall not be an identifier.

C++

An identifier can optionally be provided as a parameter name; if present in a function definition (8.4), it names a parameter (sometimes called "formal argument"). [Note: in particular, parameter names are also optional in function definitions and . . . 8.3.5p8

[Note: unused parameters need not be named. For example, 8.4p5

```
void print(int a, int)
{
    printf("a = %d\n",a);
}
```

—end note]

Source developed using a C++ translator may contain unnamed parameters, which will cause a constraint violation if processed by a C translator.

1827 If the declarator includes an identifier list, each declaration in the declaration list shall have at least one declarator, those declarators shall declare only identifiers from the identifier list, and every identifier in the identifier list shall be declared. identifier list declare at least one declarator

C90

The requirement that every identifier in the identifier list shall be declared is new in C99. In C90 undeclared identifiers defaulted to having type **int**.
Source files that translated without a diagnostic being issued by a C90 translator may now result in a diagnostic being generated by a C99 translator.

C++

The declaration list form of function definitions is not supported in C++.

An identifier declared as a typedef name shall not be redeclared as a parameter. 1828

C++

The form of function definition that this requirement applies to is not supported in C++.

---

function declaration list storage-class specifier

The declarations in the declaration list shall contain no storage-class specifier other than **register** and no initializations. 1829

C++

The form of function definition that this requirement applies to (i.e., old-style) is not supported in C++.

---

footnote 138

138) The intent is that the type category in a function definition cannot be inherited from a typedef: 1830

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

C++

The C++ Standard specifies this as a requirement in the body of the standard (8.3.5p7).

Semantics

The declarator in a function definition specifies the name of the function being defined and the identifiers of its parameters. 1831

C++

The C++ Standard does not explicitly make this association about function definitions (8.4).

If the declarator includes a parameter type list, the list also specifies the types of all the parameters; 1832

C++

If the parameter list is empty the C++ Standard defines the function as taking no arguments (8.3.5p2).

If the declarator includes an identifier list,[139] the types of the parameters shall be declared in a following declaration list. 1834

C++

The identifier list form of function definition is not supported in C++.

---

the resulting type shall be an object type. 1836

C++

The only difference, in parameter types, between a function declaration and a function definition specified by the C++ Standard is:

8.3.5p6

The type of a parameter or the return type for a function declaration that is not a definition may be an incomplete class type.

1837 If a function that accepts a variable number of arguments is defined without a parameter type list that ends with the ellipsis notation, the behavior is undefined.

C++

This C situation cannot occur in C++ because it relies on the old-style of function declaration, which is not supported in C++.

1839 Its identifier is an lvalue, which is in effect declared at the head of the compound statement that constitutes the function body (and therefore cannot be redeclared in the function body except in an enclosed block). parameter scope begins at

C++

The C++ Standard does not explicitly specify the fact that this identifier is an lvalue. However, it can be deduced from clauses 3.10p1 and 3.10p2.

The potential scope of a function parameter name in a function definition (8.4) begins at its point of declaration. If the function has a function try-block the potential scope of a parameter ends at the end of the last associated handler, else it ends at the end of the outermost block of the function definition. A parameter name shall not be redeclared in the outermost block of the function definition nor in the outermost block of any handler associated with a function try-block. 3.3.2p2

1840 The layout of the storage for parameters is unspecified.

C++

The C++ Standard does not explicitly specify any storage layout behavior for parameters.

1841 On entry to the function, the size expressions of each variably modified parameter are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment. function entry parameter type evaluated

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

1843 After all parameters have been assigned, the compound statement that constitutes the body of the function definition is executed.

C90

The C90 Standard does not explicitly specify this behavior.

C++

The C++ Standard does not explicitly specify this behavior.

1844 If the } that terminates a function is reached, and the value of the function call is used by the caller, the behavior is undefined. function termination reaching }

C++

6.6.3p2 *Flowing off the end of a function is equivalent to a* **`return`** *with no value; this results in undefined behavior in a value-returning function.*

The C++ Standard does not require that the caller use the value returned for the behavior to be undefined; this behavior can be deduced without any knowledge of the caller.

```
int f(int p_1)
{
if (p_1 > 10)
   return 2;
}

void g(void)
{
int loc = f(11);

f(2); /* does not change the conformance status of the program */
      // undefined behavior
}
```

EXAMPLE 1 In the following: 1845

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

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

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

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

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

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

C++

The second definition of `max` uses a form of function definition that is not supported in C++.

6.9.2 External object definitions

Semantics

---

object external definition

If the declaration of an identifier for an object has file scope and an initializer, the declaration is an external definition for the identifier. 1848

C++

The C++ Standard does not define the term *external definition*. The object described above is simply called a *definition* in C++.

1849 A declaration of an identifier for an object that has file scope without an initializer, and without a storage-class specifier or with the storage-class specifier **static**, constitutes a *tentative definition*. tentative definition

C++

The C++ Standard does not define the term *tentative definition*. Neither does it define a term with a similar meaning. A file scope object declaration that does not include an explicit storage-class specifier is treated, in C++, as a definition, not a tentative definition.

A translation unit containing more than one tentative definition (in C terms) will cause a C++ translator to issue a diagnostic.

```
int glob;
int glob; /* does not change the conformance status of program */
          // ill-formed program
```

1850 If a translation unit contains one or more tentative definitions for an identifier, and the translation unit contains no external definition for that identifier, then the behavior is exactly as if the translation unit contains a file scope declaration of that identifier, with the composite type as of the end of the translation unit, with an initializer equal to 0. object definition implicit

C++

The C++ Standard does not permit more than one definition in the same translation unit (3.2p1) and so does not need to specify this behavior.

1852 EXAMPLE 1 EXAMPLE linkage

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

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

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

C++

The tentative definitions are all definitions with external linkage in C++.

1853 EXAMPLE 2 If at the end of the translation unit containing EXAMPLE tentative array definition

```
int i[];
```

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

C90

This example was added to the C90 Standard by the response to DR #011.

C++

This example is ill-formed C++.

6.10 Preprocessing directives

preprocessor directives syntax

1854

```
preprocessing-file:
                group_opt
group:
                group-part
                group group-part
group-part:
                if-section
                control-line
                text-line
                # non-directive
if-section:
                if-group elif-groups_opt else-group_opt endif-line
if-group:
                # if     constant-expression new-line group_opt
                # ifdef  identifier new-line group_opt
                # ifndef identifier new-line group_opt
elif-groups:
                elif-group
                elif-groups elif-group
elif-group:
                # elif   constant-expression new-line group_opt
else-group:
                # else   new-line group_opt
endif-line:
                # endif  new-line
control-line:
                # include pp-tokens new-line
                # define  identifier replacement-list new-line
                # define  identifier lparen identifier-list_opt )
                                                  replacement-list new-line
                # define  identifier lparen ... ) replacement-list new-line
                # define  identifier lparen identifier-list , ... )
                                                  replacement-list new-line
                # undef   identifier new-line
                # line    pp-tokens new-line
                # error   pp-tokens_opt new-line
                # pragma  pp-tokens_opt new-line
                #         new-line
text-line:
                pp-tokens_opt new-line
non-directive:
                pp-tokens new-line
lparen:
                a ( character not immediately preceded by white-space
replacement-list:
                pp-tokens_opt
```

pp-tokens:

 preprocessing-token
 pp-tokens preprocessing-token

new-line:

 the new-line character

C90

Support for the following syntax is new in C99:

group-part:
 # non-directive
control-line:
 # **define** *identifier lparen* ...) *replacement-list new-line*
 # **define** *identifier lparen identifier-list* , ...)
 replacement-list new-line

The left hand side terms *text-line* and *non-directive* were introduced in C99. Their right-hand side occurred directly in the right-hand side of *group-part* in the C90 Standard.

text-line:
 pp-tokens$_{opt}$ *new-line*
non-directive:
 pp-tokens new-line

The definition of *lparen* in C90 was:

lparen:
 the left-parentheses character without preceding white-space

C++

The C++ Standard specifies the same syntax as the C90 Standard (16p1).

Description

1856 The first token in the sequence is a # preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character ~~,~~ .

C90

The C90 Standard did not contain the words “(at the start of translation phase 4)”, which were added by the response to DR #144.

C++

Like C90, the C++ Standard does not specify the start of translation phase 4.

1858 A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro. preprocessing directive ended by

C90

This wording was not in the C90 Standard, and was added by the response to DR #017.

C++

Like the original C90 Standard, the C++ Standard does not explicitly specify this behavior. However, given that vendors are likely to use a preprocessor that is identical to the one used in their C product (or the one that used to be used, if they nolonger market a C product), it is unlikely that the behaviors seen in practice will be different.

A non-directive shall not begin with any of the directive names appearing in the syntax. 1862

C90

Explicit support for *non-directive* is new in C99.

C++

Explicit support for *non-directive* is new in C99 and is not discussed in the C++ Standard.

When in a group that is skipped (6.10.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character. 1863

C90

The C90 Standard did not explicitly specify this behavior.

C++

Like C90, the C++ Standard does not explicitly specify this behavior.

Constraints

Semantics

EXAMPLE EMPTY #

EXAMPLE In: 1868

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

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

C90

This example was not in the C90 Standard and was added by the response to DR #144.

C++

This example does not appear in the C++ Standard.

6.10.1 Conditional inclusion

Constraints

Semantics

#if identifier replaced by 0

After all replacements due to macro expansion and the **defined** unary operator have been performed, all remaining identifiers (including those lexically identical to keywords) are replaced with the pp-number **0**, and then each preprocessing token is converted into a token. 1878

C++

In the C++ Standard **true** and **false** are not identifiers (macro names), they are literals:

16.1p4 *. . . , except for **true** and **false**, are replaced with the pp-number 0, . . .*

If the character sequence `true` is not defined as a macro and appears within the constant-expression of a conditional inclusion directive, when preprocessed by a C++ translator this character sequence will be treated as having the value one, not zero.

1880 For the purposes of this token conversion and evaluation, all signed integer types and all unsigned integer types act as if they have the same representation as, respectively, the types **`intmax_t`** and **`uintmax_t`** defined in the header **`<stdint.h>`**.[142)]

#if operand type uintmax_*

C90

> *The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 6.4 using arithmetic that has at least the ranges specified in 5.2.4.2, except that* **`int`** *and* **`unsigned int`** *act as if they have the same representation as, respectively,* **`long`** *and* **`unsigned long`**.

The ranks of the integer types used for the operands of the controlling constant expression differ between C90 and C99 (although in both cases the rank is the largest that an implementation is required to support). Those cases where the value of the operand exceeded the representable range in C90 (invariably resulting in the value wrapping) are likely to generate a very large value in C99.

C++

The C++ Standard specifies the same behavior as C90 (see the C90 subsection above).

6.10.2 Source file inclusion

Constraints

Semantics

1909 The implementation shall provide unique mappings for sequences consisting of one or more ~~letters or digits (as defined in 5.2.1)~~ nondigits or digits (6.4.2.1) followed by a period (**`.`**) and a single ~~letter~~ nondigit.

#include mapping to host file

C++

> *The implementation provides unique mappings for sequences consisting of one or more nondigits (2.10) followed by a period (.) and a single nondigit.*

16.2p5

1910 The first character shall ~~be a letter~~ not be a digit.

C90

The requirement that the first character not be a digit is new in C99. Given that it is more restrictive than that required for existing C90 implementations (and thus existing code) it is unlikely that existing code will be affected by this requirement.

C++

This requirement is new in C99 and is not specified in the C++ Standard (the argument given in the C90 subsection (above) also applies to C++).

1911 The implementation may ignore the distinctions of alphabetical case and restrict the mapping to eight significant characters before the period.

header name significant characters

C90

The limit specified by the C90 Standard was six significant characters. However, implementations invariably used the number of significant characters available in the host file system (i.e., they do not artificially limit the number of significant characters). It is unlikely that a header of source file will fail to be identified because of a difference in what used to be a non-significant character.

C++

The C++ Standard does not give implementations any permissions to restrict the number of significant characters before the period (16.1p5). However, the limits of the file system used during translation are likely to be the same for both C and C++ implementations and consequently no difference is listed here.

6.10.3 Macro replacement

Constraints

object-like macro redefinition

An identifier currently defined as an object-like macro shall not be redefined by another **#define** preprocessing directive unless the second definition is an object-like macro definition and the two replacement lists are identical. 1919

C90

The wording in the C90 Standard was modified by the response to DR #089.

There shall be white-space between the identifier and the replacement list in the definition of an object-like macro. 1921

C90

The response to DR #027 added the following requirements to the C90 Standard.

DR #027

Correction

Add to subclause 6.8, page 86 (Constraints):

*In the definition of an object-like macro, if the first character of a replacement list is not a character required by subclause 5.2.1, then there shall be white-space separation between the identifier and the replacement list.**

*[Footnote *: This allows an implementation to choose to interpret the directive:*

```
#define THIS$AND$THAT(a, b) ((a) + (b))
```

as defining a function-like macro THISANDTHAT, rather than an object-like macro THIS. Whichever choice it makes, it must also issue a diagnostic.]

However, the complex interaction between this specification and UCNs was debated during the C9X review process and it was decided to simplify the requirements to the current C99 form.

```
#define TEN.1 /* Define the macro TEN to have the body .1 in C90. */
              /* A constraint violation in C99. */
```

C++

The C++ Standard specifies the same behavior as the C90 Standard.

If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. 1922

C90

If (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined.

The behavior of the following was discussed in DR #003q3, DR #153, and raised against C99 in DR #259 (no committee response was felt necessary).

```
#define foo() A
#define bar(B) B

foo() // no arguments
bar() // one empty argument?
```

What was undefined behavior in C90 (an empty argument) is now explicitly supported in C99. The two most likely C90 translator undefined behaviors are either to support them (existing source developed using such a translator will may contain empty arguments in a macro invocation), or to issue a diagnostic (existing source developed using such a translator will not contain any empty arguments in a macro invocation).

C++

The C++ Standard contains the same wording as the C90 Standard.

C++ translators are not required to correctly process source containing macro invocations having any empty arguments.

1923 Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the `...`). ... arguments macro

C90

Support for the form `...` is new in C99.

C++

Support for the form `...` is new in C99 and is not specified in the C++ Standard.

1925 The identifier `__VA_ARGS__` shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the ~~arguments~~parameters.

C90

Support for `__VA_ARGS__` is new in C99.

Source code declaring an identifier with the spelling `__VA_ARGS__` will cause a C99 translator to issue a diagnostic (the behavior was undefined in C90).

C++

Support for `__VA_ARGS__` is new in C99 and is not specified in the C++ Standard.

Semantics

1933 A preprocessing directive of the form macro function-like

**define** *identifier lparen identifier-list*$_{opt}$ **)** *replacement-list new-line*
**define** *identifier lparen* **...** **)** *replacement-list new-line*
**define** *identifier lparen identifier-list* **,** **...** **)** *replacement-list new-line*

defines a *function-like macro* with ~~arguments,~~ parameters, whose use is similar syntactically to a function call.

C90

Support for the `...` notation in function-like macro definitions is new in C99.

C++

Support for the `...` notation in function-like macro definitions is new in C99 and is not specified in the C++ Standard.

1941 If there is a `...` in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the *variable arguments*.

C90

Support for `...` in function-like macro definitions is new in C99.

C++

Support for `...` in function-like macro definitions is new in C99 and is not specified in the C++ Standard.

6.10.3.1 Argument substitution

---

argument substitution

After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. 1945

C++

C++ does not support empty arguments___? .E_COMMENT

An identifier `__VA_ARGS__` that occurs in the replacement list shall be treated as if it were a parameter, and the variable arguments shall form the preprocessing tokens used to replace it. 1949

C90

Support for `__VA_ARGS__` is new in C99.

C++

Support for `__VA_ARGS__` is new in C99 and is not specified in the C++ Standard.

6.10.3.2 The # operator

Constraints

Semantics

---

escape sequence handling

Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character constants: a \ character is inserted before each " and \ character of a character constant or string literal (including the delimiting " characters), except that it is implementation-defined whether a \ character is inserted before the \ character beginning a universal character name. 1954

C90

Support for universal character names is new in C99.

C++

Support for universal character names is available in C++. However, wording for this clause of the C++ Standard was copied from C90, which did not support universal character names. The behavior of a C++ translator can be viewed as being equivalent to another C99 translator, in this regard. A C++ translator is not required to document its handling of a \ character before a universal character name.

The character string literal corresponding to an empty argument is "". 1956

C90

An occurrence of an empty argument in C90 caused undefined behavior.

C++

Like C90, the behavior in C++ is not explicitly defined (some implementations e.g., Microsoft C++, do not support empty arguments).

6.10.3.3 The ## operator

Constraints

Semantics

1960 however, if an argument consists of no preprocessing tokens, the parameter is replaced by a *placemarker* preprocessing token instead.[148)]

argument no tokens replaced by placemarker token

C90

The explicitly using the concept of a placemarker preprocessing token is new in C99.

C++

The explicit concept of a placemarker preprocessing token is new in C99 and is not described in C++.

1962 Placemarker preprocessing tokens are handled specially: concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token.

placemarker preprocessor

C90

The concept of placemarker preprocessing tokens is new in the C99 Standard. The behavior of concatenating an empty argument with preprocessing token was not explicitly defined in C90, it was undefined behavior.

C++

Like C90, the behavior of concatenating an empty argument with preprocessing token is not explicitly defined in C++, it is undefined behavior.

1966 EXAMPLE In the following fragment:

EXAMPLE # ## #

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)

char p[] = join(x, y); // equivalent to
                       // char p[] = "x ## y"
```

The expansion produces, at various stages:

```
join(x, y)

in_between(x hash_hash y)

in_between(x ## y)

mkstr(x ## y)

"x ## y"
```

In other words, expanding **hash_hash** produces a new token, consisting of two adjacent sharp signs, but this new token is not the ## operator.

C++

This example is the response to a DR against C90. While there has been no such DR in C++, it is to be expected that WG21 would provide the same response.

1967 148) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

footnote 148

C90

Support for the concept of placemarker preprocessing tokens is new in C99.

C++

Support for the concept of placemarker preprocessing tokens is new in C99 and they are not described in the C++ Standard.

6.10.3.4 Rescanning and further replacement

---

rescanning

After all parameters in the replacement list have been substituted and # and ## processing has taken place, all placemarker preprocessing tokens are removed. 1968

C90

Support for the concept of placemarker preprocessing tokens is new in C99.

C++

Support for the concept of placemarker preprocessing tokens is new in C99 and does not exist in C++.

---

expanded token sequence not treated as a directive

The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 6.10.9 below. 1973

C90

Support for **_Pragma** unary operator expressions is new in C99.

C++

Support for **_Pragma** unary operator expressions is new in C99 and is not available in C++.

6.10.3.5 Scope of macro definitions

---

macro definition no significance after

Macro definitions have no significance after translation phase 4. 1975

C90

This observation is new in C99.

C++

This observation is not made in the C++ document.

EXAMPLE placemarker

EXAMPLE 5 To illustrate the rules for placemarker preprocessing tokens, the sequence 1982

```
#define t(x,y,z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,,7), t(8,9,),
           t(10,,), t(,11,), t(,,12), t(,,) };
```

results in

```
int j[] = { 123, 45, 67, 89,
            10, 11, 12,  };
```

C90

This example is new in the C99 Standard and contains undefined behavior in C90.

C++

The C++ Standard specification is the same as that in the C90 Standard,

---

EXAMPLE variable macro arguments

EXAMPLE 7 Finally, to show the variable argument list macro facilities: 1984

```
#define debug(...)       fprintf(stderr, __VA_ARGS__)
#define showlist(...)       puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                printf(__VA_ARGS__))
```

```
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is %d", x, y);
```

results in

```
fprintf(stderr,  "Flag" );
fprintf(stderr,  "X = %d\n", x );
puts( "The first, second, and third items." );
((x>y)?puts("x>y"):
                printf("x is %d but y is %d", x, y));
```

C90

Support for macros taking a variable number of arguments is new in C99.

C++

Support for macros taking a variable number of arguments is new in C99 and is not supported in C++.

6.10.4 Line control

Constraints

Semantics

1988 The digit sequence shall not specify zero, nor a number greater than 2147483647.

C90

The limit specified in the C90 Standard was 32767.

C++

Like C90, the limit specified in the C++ Standard is 32767.

1992 The directive resulting after all replacements shall match one of the two previous forms and is then processed as appropriate.

C++

The C++ Standard uses different wording that has the same meaning.

> *If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.* 16.4p5

6.10.5 Error directive

Semantics

1993 A preprocessing directive of the form #error

**error** *pp-tokens*$_{opt}$ *new-line*

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens.

C++

16.5p1

. . . , and renders the program ill-formed.

Both language standards require that a diagnostic be issued. But the C Standard does not specify that the construct alters the conformance status of the translation unit. However, given that the occurrence of this directive causes translation to terminate, this is a moot point.

#error 89 terminate translation

6.10.6 Pragma directive

Semantics

#pragma directive

A preprocessing directive of the form 1994

```
# pragma pp-tokens_opt new-line
```

where the preprocessing token **STDC** does not immediately follow **pragma** in the directive (prior to any macro replacement)[149] causes the implementation to behave in an implementation-defined manner.

C90

The exception for the preprocessing token **STDC** is new in C99.

C++

The exception for the preprocessing token **STDC** is new in C99 and is not specified in C++.

The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. 1995

C90

These possibilities were not explicitly specified in the C90 Standard.

C++

These possibilities are not explicitly specified in the C++ Standard.

If the preprocessing token **STDC** does immediately follow **pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms[150] whose meanings are described elsewhere: 1997

```
#pragma STDC FP_CONTRACT on-off-switch
#pragma STDC FENV_ACCESS on-off-switch
#pragma STDC CX_LIMITED_RANGE on-off-switch
on-off-switch:  one of
                ON        OFF        DEFAULT
```

C90

Support for the preprocessing token **STDC** in **pragma** directives is new in C99.

C++

Support for the preprocessing token **STDC** in **pragma** directives is new in C99 and is not specified in the C++ Standard.

---

footnote 149

149) An implementation is not required to perform macro replacement in pragmas, but it is permitted except for in standard pragmas (where **STDC** immediately follows **pragma**). 1999

C90

This footnote is new in C99.

C++

This footnote is new in C99 and is not specified in the C++ Standard.

2000 If the result of macro replacement in a non-standard pragma has the same form as a standard pragma, the behavior is still implementation-defined;

C90

Support for standard pragmas is new in C99.

C++

Support for standard pragmas is new in C99 and is not specified in the C++ Standard.

6.10.7 Null directive

Semantics

6.10.8 Predefined macro names

2009 `__STDC__` The integer constant `1`, intended to indicate a conforming implementation. __STDC__ macro

C++

> *Whether __STDC__ is predefined and if so, what its value is, are implementation-defined.* 16.8p1

It is to be expected that a C++ translator will not define the `__STDC__`, the two languages are different, although a conforming C++ translator may often behave in a fashion expected of a conforming C translator. Some C++ translators have a switch that causes them to operate in a C compatibility mode (in this case it is to be expected that this macro will be defined as per the requirements of the C Standard).

2010 `__STDC_HOSTED__` The integer constant `1` if the implementation is a hosted implementation or the integer constant `0` if it is not. __STDC_HOSTED__ macro

C90

Support for the `__STDC_HOSTED__` macro is new in C99.

C++

Support for the `__STDC_HOSTED__` macro is new in C99 and it is not available in C++.

2011 `__STDC_VERSION__` The integer constant `199901L`.[153] __STDC_VERSION__ macro

C90

Support for the `__STDC_VERSION__` macro was first introduced in Amendment 1 to C90, where it was specified to have the value `199409L`. In a C90 implementation (with no support for Amendment 1) occurrences of this macro are likely to be replaced by 0 (because it will not be defined as a macro). 1878 #if identifier replaced by 0

C++

Support for the `__STDC_VERSION__` macro is not available in C++.

2014 The following macro names are conditionally defined by the implementation:

C90

Support for conditionally defined macros is new in C99.

C++

Support for conditionally defined macros is new in C99 and none are defined in the C++ Standard.

__STDC_IEC_559__ macro

__STDC_IEC_559__ The integer constant **1**, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic). 2015

C90

Support for the __STDC_IEC_559__ macro is new in C99.

C++

Support for the __STDC_IEC_559__ macro is new in C99 and it is not available in C++.

The C++ Standard defines, in the std namespace:

18.2.1.1

```
static const bool is_iec559 = false;
```

false is the default value. In the case where the value is **true** the requirements stated in C99 also occur in the C++ Standard. The member is_iec559 is part of the numerics template and applies on a per type basis. However, the requirement for the same value representation, of floating types, implies that all floating types are likely to have the same value for this member.

footnote 152

152) The presumed source file name and line number can be changed by the **#line** directive. 2017

C90

This observation is new in the C99 Standard.

C++

Like C90, the C++ Standard does not make this observation.

__STDC__IEC_559_COMPLEX__ macro

__STDC_IEC_559_COMPLEX__ The integer constant **1**, intended to indicate adherence to the specifications in informative annex G (IEC 60559 compatible complex arithmetic). 2020

C90

Support for the __STDC_IEC_559_COMPLEX__ macro is new in C99.

C++

Support for the __STDC_IEC_559_COMPLEX__ macro is new in C99 and is not available in C++.

__STDC_ISO_10646__ macro

__STDC_ISO_10646__ An integer constant of the form **yyyymmL** (for example, **199712L**) ~~,~~. 2021

C90

Support for the __STDC_ISO_10646__ macro is new in C99.

C++

Support for the __STDC_ISO_10646__ macro is new in C99 and is not available in C++.

If this symbol is defined, then every character in the Unicode required set, when stored in an object of type **wchar_t**, has the same value as the short identifier of that character. 2022

C90

This form of encoding was not mentioned in the C90 Standard.

C++

This form of encoding is not mentioned in the C++ Standard.

predefined macros not #defined

None of these macro names, nor the identifier **defined**, shall be the subject of a **#define** or a **#undef** preprocessing directive. 2026

C++

The C++ Standard uses different wording that has the same meaning.

> *If any of the pre-defined macro names in this subclause, or the identifier* **`defined`**, *is the subject of a* **`#define`** *or a* **`#undef`** *preprocessing directive, the behavior is undefined.* 16.8p3

2027 Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore. macro name predefined reserved

C++

The C++ Standard does not reserve names for any other predefined macros.

2028 The implementation shall not predefine the macro **`__cplusplus`**, nor shall it define it in any standard header. __cplusplus

C90

This requirement was not specified in the C90 Standard. Given the prevalence of C++ translators, vendors were aware of the issues involved in predefining such a macro name (i.e., they did not do it).

C++

> *The name __cplusplus is defined to the value 199711L when compiling a C++ translation unit.*[143)] 16.8p1

6.10.9 Pragma operator

Semantics

2030 A unary operator expression of the form: _Pragma operator

```
_Pragma ( string-literal )
```

is processed as follows: The string literal is *destringized* by deleting the **`L`** prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence `\"` by a double-quote, and replacing each escape sequence `\\` by a single backslash.

C90

Support for the **`_Pragma`** unary operator is new in C99.

C++

Support for the **`_Pragma`** unary operator is new in C99 and it is not available in C++.

6.11 Future language directions

6.11.1 Floating types

2034 Future standardization may include additional floating-point types, including those with greater range, precision, or both than **`long double`**. floating types future language directions

C90

This future direction is new in C99.

C++

The C++ Standard specifies (Annex D) deprecated features. With one exception these all relate to constructs specific to C++.

D.5p2 *Each C header, whose name has the form `name.h`, behaves as if each name placed in the Standard library namespace by the corresponding `cname` header is also placed within the namespace scope of the namespace `std` and is followed by an explicit `using-declaration` (7.3.3)*

6.11.2 Linkages of identifiers

identifier linkage future language directions

Declaring an identifier with internal linkage at file scope without the **`static`** storage-class specifier is an obsolescent feature. 2035

C90

This future direction is new in C99.

6.11.3 External names

significant characters future language directions

Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations. 2036

C90

Part of the future language direction specified in C90 was implemented in C99.

Restriction of the significance of an external name to fewer than 31 characters or to only one case is an obsolescent feature that is a concession to existing implementations.

6.11.4 Character escape sequences

6.11.5 Storage-class specifiers

6.11.6 Function declarators

6.11.7 Function definitions

6.11.8 Pragma directives

Pragma directives future language directions

Pragmas whose first preprocessing token is **`STDC`** are reserved for future standardization. 2042

C90

Support for this form of **`pragma`** directive is new in C99.

6.11.9 Predefined macro names

Predefined macro names future language directions

Macro names beginning with **`__STDC_`** are reserved for future standardization. 2043

C90

The specification of this reserved set of macro name spellings is new in C99.

References

1. T. H. Gibbs. *The design and implementation of a parser and front-end for the ISO C++ language and validation of the parser*. PhD thesis, Clemson University, May 2003.
2. ISO. *ISO/IEC 9945-1:1990 Information technology —Portable Operating System Interface (POSIX)*. ISO, 1990.
3. D. M. Ritchie. The development of the C language. *Second History of Programming Languages conference*, 1993.
4. B. Stroustrup. *The Design and Evolution of C++*. Addison–Wesley, 1999.
5. R. L. Velduizen. C++ templates as partial evaluation. Technical Report TR519, Indiana University, July 2000.
6. E. D. Willink. *Meta-Compilation for C++*. PhD thesis, University of Surrey, June 2001.