# The New C Standard (Excerpted material)

## An Economic and Cultural Commentary

**Derek M. Jones**
derek@knosof.co.uk

# 4. Conformance{ref conformance}

### Commentary

In the C90 Standard this header was titled *Compliance*. Since this standard talks about conforming and strictly conforming programs it makes sense to change this title. Also, from existing practice, the term *Conformance* is used by voluntary standards, such as International Standards, while the term *Compliance* is used by involuntary standards, such as regulations and laws.

SC22 had a Working Group responsible for conformity and validation issues, WG12. This WG was formed in 1983 and disbanded in 1989. It produced two documents: *ISO/IEC TR 9547:1988— Test methods for programming language processors – guidelines for their development and procedures for their approval* and *ISO/IEC TR 10034:1990— Guidelines for the preparation of conformity clauses in programming language standards*.

---

shall

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

### Commentary

How do we know which is which? In many cases the context in which the *shall* occurs provides the necessary information. Most usages of *shall* apply to programs and these commentary clauses only point out those cases where it applies to implementations.

shall 84 outside constraint constraint

The extent to which implementations are required to follow the requirements specified using *shall* is affected by the kind of subclause the word appears in. Violating a *shall* requirement that appears inside a subsection headed *Constraint* clause is a constraint violation. A conforming implementation is required to issue a diagnostic when it encounters a violation of these constraints.

The term *should* is not defined by the standard. This word only appears in footnotes, examples, recommended practices, and in a few places in the library. The term *must* is not defined by the standard and only occurs once in it as a word.

EXAMPLE compatible function prototypes

### C++

ISO 84 shall rules

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.

### Coding Guidelines

Coding guidelines are best phrased using "shall" and by not using the words "should", "must", or "may".

### Usage

The word *shall* occurs 537 times (excluding occurrences of *shall not*) in the C Standard.

---

conversely, "shall not" is to be interpreted as a prohibition. 83

### Commentary

shall 84 outside constraint constraint

In some cases this prohibition requires a diagnostic to be issued and in others it results in undefined behavior. An occurrence of a construct that is the subject of a *shall not* requirement that appears inside a subsection headed *Constraint* clause is a constraint violation. A conforming implementation is required to issue a diagnostic when it encounters a violation of these constraints.

### Coding Guidelines

Coding guidelines are best phrased using *shall not* and by not using the words *should not*, *must not*, or *may not*.

### Usage

The phrase *shall not* occurs 51 times (this includes two occurrences in footnotes) in the C Standard.

---

shall outside constraint

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

### Commentary

ISO shall rules

This C sentence brings us onto the use of ISO terminology and the history of the C Standard. ISO use of terminology requires that the word *shall* implies a constraint, irrespective of the subclause it appears in. So under ISO rules, all sentences that use the word *shall* represent constraints. But the C Standard was first published as an ANSI standard, ANSI X3.159-1989. It was adopted by ISO, as ISO/IEC 9899:1990, the following year with minor changes (e.g., the term Standard was replaced by International Standard and there was a slight renumbering of the major clauses; there is a `sed` script that can convert the ANSI text to the ISO text), but the *shall*s remained unchanged.

If you, dear reader, are familiar with the ISO rules on *shall*, you need to forget them when reading the C Standard. This standard defines its own concept of constraints and meaning of *shall*.

### 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.

84 ISO shall rules

### Coding Guidelines

Many developers are not aware that the C Standard's meaning of the term *shall* is context-dependent. If developers have access to a copy of the C Standard, it is important that this difference be brought to their attention; otherwise, there is the danger that they will gain false confidence in thinking that a translator will issue a diagnostic for all violations of the stated requirements. In a broader sense educating developers about the usage of this term is part of their general education on conformance issues.

### Usage

The word *shall* appears 454 times outside of a Constraint clause; however, annex J.2 only lists 190 undefined behaviors. The other uses of the word *shall* apply to requirements on implementations, not programs.

---

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

### Commentary

Failure to find an explicit definition of behavior could, of course, be because the reader did not look hard enough. Or it could be because there was nothing to find, implicitly undefined behavior. On the whole the Committee does not seem to have made any obvious omissions of definitions of behavior. Those DRs that have been submitted to WG14, which have later turned out to be implicitly undefined behavior, have involved rather convoluted constructions. This specification for the omissions of an explicit definition is more of a catch-all rather than an intent to minimize wording in the standard (although your author has heard some Committee members express the view that it was never the intent to specify every detail).

The term *shall* can also mean undefined behavior.

84 shall outside constraint

### C++

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

### Coding Guidelines

Is it worth highlighting omissions of explicit definitions of behavior in coding guidelines (the DRs in the record of response log kept by WG14 provides a confirmed source of such information)? Pointing out that the C Standard does not always fully define a construct may undermine developers' confidence in it, resulting in them claiming that a behavior was undefined because they could find no mention of it in the standard when a more thorough search would have located the necessary information.

### Example

The following quote is from Defect Report #017, Question 19 (raised against C90).

*X3J11 previously said, "The behavior in this case could have been specified, but the Committee has decided more than once not to do so. [They] do not wish to promote this sort of macro replacement usage." I interpret this as saying, in other words, "If we don't define the behavior nobody will use it." Does anybody think this position is unusual?*

*Response*

*If a fully expanded macro replacement list contains a function-like macro name as its last preprocessing token, it is unspecified whether this macro name may be subsequently replaced. If the behavior of the program depends upon this unspecified behavior, then the behavior is undefined.*

*For example, given the definitions:*

```
#define f(a) a*g

#define g(a) f(a)
```

*the invocation:*

```
f(2)(9)
```

*results in undefined behavior. Among the possible behaviors are the generation of the preprocessing tokens:*

```
2*f(9)
```

*and*

```
2*9*g
```

```
Correction
Add to subclause G.2, page 202:
-- A fully expanded macro replacement list contains a
function-like macro name as its last preprocessing token (6.8.3).
```

Subclause G.2 was the C90 annex listing undefined behavior. Different wording, same meaning, appears in annex J.2 of C99.

---

There is no difference in emphasis among these three; 86

**Commentary**

It is not possible to write a construct whose behavior is more undefined than another construct, simply because of the wording used, or not used, in the standard.

**Coding Guidelines**

There is nothing to be gained by having coding guideline documents distinguish between the different ways undefined behavior is indicated in the C Standard.

---

they all describe "behavior that is undefined". 87

---

correct program

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. 88

**Commentary**

unspecified behavior

As pointed out elsewhere, any nontrivial program will contain unspecified behavior.

A wide variety of terms are used by developers to refer to programs that are not correct. The C Standard does not define any term for this kind of program.

Terms, such as *fault* and *defect*, are defined by various standards:

*defect. See fault.*

*error. (1) A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretical correct value or condition.*

*(2) Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, incorrect translation or omission of a requirement in the design specification. This is not the preferred usage.*

*fault. (1) An accidental condition that causes a functional unit to fail to perform its required function.*

*(2) A manifestation of an error(2) in software. A fault, if encountered, may cause a failure. Synonymous with bug.*

ANSI/AIAA R–013-1992, Recommended Practice for Software Reliability

*Error (1) A discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. (2) Human action that results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, and incorrect translation or omission of a requirement in the design specification. This is not a preferred usage.*

*Failure (1) The inability of a system or system component to perform a required function with specified limits. A failure may be produced when a fault is encountered and a loss of the expected service to the user results. (2) The termination of the ability of a functional unit to perform its required function. (3) A departure of program operation from program requirements.*

*Failure Rate (1) The ratio of the number of failures of a given category or severity to a given period of time; for example, failures per month. Synonymous with failure intensity. (2) The ratio of the number of failures to a given unit of measure; for example, failures per unit of time, failures per number of transactions, failures per number of computer runs.*

*Fault (1) A defect in the code that can be the cause of one or more failures. (2) An accidental condition that causes a functional unit to fail to perform its required function. Synonymous with bug.*

*Quality The totality of features and characteristics of a product or service that bears on its ability to satisfy given needs.*

*Software Quality (1) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, to conform to specifications. (2) The degree to which software possesses a desired combination of attributes. (3) The degree to which a customer or user perceives that software meets his or her composite expectations. (4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.*

*Software Reliability (1) The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system, as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. (2) The ability of a program to perform a required function under stated conditions for a stated period of time.*

**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++**

*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:*

1.4p2

*— 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++.

### Common Implementations

A program's source code may look correct when mentally executed by a developer. The standard assumes that C programs are correctly translated. Translators are programs like any other, they contain faults. Until the 1990s, the idea of proving the correctness of a translator for a commercially used language was not taken seriously. The complexity of a translator and the volume of source it contained meant that the resources required would be uneconomical. Proofs that were created applied to toy languages, or languages that were so heavily subseted as to be unusable in commercial applications.

Having translators generate correct machine code continues to be very important. Processors continue to become more powerful and support gigabytes of main storage. Researchers continue to increase the size of the language subsets for which translators have been proved correct.[7, 9, 11] They have also looked at proving some of the components of an existing translator, `gcc`, correct.[8]

### Coding Guidelines

The phrase *the program is correct* is used by developers in a number of different contexts, for instance, to designate intended program behavior, or a program that does not contain faults. When describing adherence to the requirements of the C Standard, the appropriate term to use is *conformance*.

Adhering to coding guidelines does not guarantee that a program is correct. The phrase *correct program* does not really belong in a coding guidelines document. These coding guidelines are silent on the issue of what constitutes correct data.

---

#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

### Commentary

The intent is to provide a mechanism to unconditionally cause translation to fail. Prior to this explicit requirement, it was not guaranteed that a **`#error`** directive would cause translation to fail, if encountered, although in most cases it did.

#error

### 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).

### Common Implementations

Most, but not all, C90 implementations do not successfully translate a preprocessing translation unit containing this directive (unless skipping an arm of a conditional inclusion). Some K&R implementations failed to translate any source file containing this directive, no matter where it occurred. One solution to this problem is to write the source as `??=error`, because a K&R compiler would not recognize the trigraph.

Some implementations include support for a `#warning` preprocessor directive, which causes a diagnostic to be issued without causing translation to fail. #warning

### Example

```
#if CHAR_BIT != 8
#error Networking code requires byte == octet
#endif
```

---

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

### Commentary

In other words, a strictly conforming program cannot use extensions, either to the language or the library. A strictly conforming program is intended to be maximally portable and can be translated and executed by any conforming implementation. Nothing is said about using libraries specified by other standards. As far as the translator is concerned, these are translation units processed in translation phase 8. There is no way of telling apart user-written translation units and those written by third parties to conform to another API standard. translation phase 8

The Standard does not forbid extensions provided that they do not invalidate strictly conforming programs, and the translator must allow extensions to be disabled as discussed in Rationale §4. Otherwise, extensions to a conforming implementation lie in such realms as defining semantics for syntax to which no semantics is ascribed by the Standard, or giving meaning to undefined behavior. Rationale

### C++

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

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 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.

*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).* 1.4p7

### Other Languages

Most language specifications do not have as sophisticated a conformance model as C.

### Common Implementations

All implementations known to your author will successfully translate some programs that are not strictly conforming.

### Coding Guidelines

extensions 95.1 cost/benefit

This part of the definition of strict conformance mirrors the guideline recommendation on using extensions. Translating a program using several different translators, targeting different host operating systems and processors, is often a good approximation to *all implementations* (this is a tip, not a guideline recommendation).

---

strictly conforming program output shall not

It shall not produce output dependent on any unspecified, undefined, or implementation-defined behavior, and shall not exceed any minimum implementation limit. 91

### Commentary

The key phrase here is *output*. Constructs that do not affect the output of a program do not affect its conformance status (although a program whose source contains violations of constraint or syntax will never get to the stage of being able to produce any output). A translator is not required to deduce whether a construct affects the output while performing a translation. Violations of syntax and constraints must be diagnosed independent of whether the construct is ever executed, at execution time, or affects program output. These are extremely tough requirements to meet. Even the source code of some C validation suites did not meet these requirements in some cases.[5]

implementation validation 92

### Coding Guidelines

Many coding guideline documents take a strong line on insisting that programs not contain any occurrence of unspecified, undefined, or implementation-defined behaviors. As previously discussed, this is completely unrealistic for unspecified behavior. For some constructs exhibiting implementation-defined behavior, a strong case can be made for allowing their use. The issues involved in the use of constructs whose behavior is implementation-defined is discussed in the relevant sentences.

unspecified behavior

implementation-defined behavior

The issue of programs exceeding minimum implementation limits is rarely considered as being important. This is partly based on developers' lack of experience of having programs fail to translate because they exceed the kinds of limits specified in the C Standard. Program termination at execution time because of a lack of some resource is often considered to be an application domain, or program implementation issue. These coding guidelines are not intended to cover this kind of situation, although some higher-level, application-specific guidelines might.

redundant code

The issue of code that does not affect program output is discussed elsewhere.

Cg 91.1

All of a programs translated source code shall be assumed to affect its output, when determining its conformance status.

---

implementation two forms

The two forms of *conforming implementation* are hosted and freestanding. 92

### Commentary

Not all hardware containing a processor can support a C translator. For instance, a coffee machine. In these cases programs are translated on one host and executed on a completely different one. Desktop and minicomputer-based developers are not usually aware of this distinction. Their programs are usually designed to execute on hosts similar to those that translate them (same processor family and same kind of operating system).

A freestanding environment is often referred to as the *target* environment; the thinking being that source code is translated in one environment with the aim of executing it on another, the target. This terminology is only used for a hosted environment, where the program executes in a different environment from the one in which it was translated.

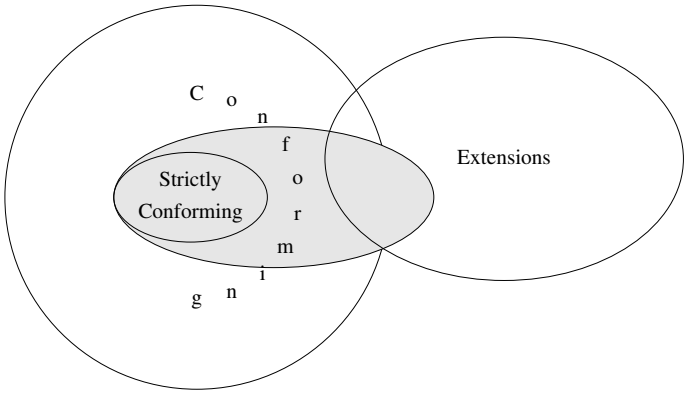


**Figure 92.1:** A conforming implementation (gray area) correctly handles all strictly conforming programs, may successfully translate and execute some of the possible conforming programs, and may include some of the possible extensions.

implementation validation

The concept of implementation-conformance to the standard is widely discussed by developers. In practice implementations are not perfect (i.e., they contain bugs) and so can never be said to be conforming. The testing of products for conformance to International Standards is a job carried out by various national testing laboratories. Several of these testing laboratories used to be involved in testing software, including the C90 language standard (language implementation validation did not prove commercially viable and there are no longer any national testing laboratories offering this service). A suite of test programs was used to measure an implementation's handling of various constructs. An implementation that successfully processed the tests was not certified to be a conforming implementation but rather (in BSI's case): "This is to certify that the language processor identified below has been found to contain no errors when tested with the identified validation suite, and is therefore deemed to conform to the language standard."

Ideally, a validation suite should have the following properties:

- Check all the requirements of the standard.
- Tests should give the same results across all implementations (they should be strictly conforming programs).
- Should not contain coding bugs.
- Should contain a test harness that enables the entire suite to be compiled/linked/executed and a pass/fail result obtained.
- Should contain a document that explains the process by which the above requirements were checked for correctness.

There are two validation suites that are widely used commercially: Perennial CVSA (version 8.1) consists of approximately 61,000 test cases in 1,430,000 lines of source code, and Plum Hall validation suite (CV-Suite 2003a) for C contains 84,546 test cases in 157,000 lines of source. A study by Jones[5] investigated the completeness and correctness of the ACVS. Ciechanowicz[2] did the same for the Pascal validation suite.

Most formal validation concentrates on language syntax and semantics. Some vendors also offer automated expression generators for checking the correctness of the generated machine code (by generating various combinations of operators and operands whose evaluation delivers a known result, which is checked by translating and executing the generated program). Wichmann[10] describes experiences using one such generator.

### Other Languages

Most other standardized languages are targeted at a hosted environment.

Some language specifications support different levels of conformance to the standard. For instance, Cobol has three implementation levels, as does SQL (Entry, Intermediate, and Full). In the case of Cobol and

Fortran, this approach was needed because of the technical problems associated with implementing the full language on the hosts of the day (which often had less memory and processing power than modern hand calculators).

The Ada language committee took the validation of translators seriously enough to produce a standard: ISO/IEC 18009:1999 *Information technology— Programming languages – Ada: Conformity assessment of a language processor*. This standard defines terms, and specifies the procedures and processes that should be followed. An Ada Conformity Assessment Test suite is assumed to exist, but nothing is said about the attributes of such a suite.

The POSIX Committee, SC22/WG15, also defined a standard for measuring conformance to its specifications. In this case they[3] attempted to provide a detailed top-level specification of the tests that needed to be performed. Work on this conformance standard was hampered by the small number of people, with sufficient expertise, willing to spend time writing it. Experience also showed that vendors producing POSIX test suites tended to write to the requirements in the conformance standard, not the POSIX standard. Lack of resources needed to update the conformance standard has meant that POSIX testing has become fossilized.

Java was originally designed to run in what is essentially a freestanding environment.

### Common Implementations

The extensive common ground that exists between different hosted implementations does not generally exist within freestanding implementations. In many cases programs intended to be executed in a hosted environment are also translated in that environment. Programs intended for a freestanding environment are rarely translated in that environment.

---

conforming hosted implementation

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

### Commentary

translation limits
implementation extensions 95
strictly conforming program 90
use features of language/library

This is a requirement on the implementation. Another requirement on the implementation deals with limits. This requirement does not prohibit an implementation from accepting programs that are not strictly conforming.

A strictly conforming program can use any feature of the language or library. This requirement is stating that a conforming hosted implementation shall implement the entire language and library, as defined by the standard (modulo those constructs that are conditional).

### C++

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

### Example

Is a conforming hosted implementation required to translate the following translation unit?

```
1  int array1[5];
2  int array2[5];
3  int *p1 = &array1[0];
4  int *p2 = &array2[0];
5
6  int DR_109()
7  {
8  return (p1 > p2);
9  }
```

relational pointer comparison
undefined if not same object

It would appear that the pointers `p1` and `p2` do not point into the same object, and that their appearance as operands of a relational operator results in undefined behavior. However, a translator would need to be certain that the function `f_1` is called, that `p1` and `p2` do not point into the same object, and that the output of any program that calls it is dependent on it. Even in the case:

```
1  int f_2(void)
2  {
```

```
3   return 1/0;
4   }
```

a translator cannot fail to translate the translation unit unless it is certain that the function `f_2` is called.

---

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

**Commentary**

This is a requirement on the implementation. There is nothing to prevent a conforming implementation supporting additional standard headers, that are not listed here.

Complex types were added to help the Fortran supercomputing community migrate to C. They are very unlikely to be needed in a freestanding environment.

The standard headers that are required to be supported define macros, typedefs, and objects only. The runtime library support needed for them is therefore minimal. The header **<stdarg.h>** is the only one that may need runtime support.

**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.

**Common Implementations**

String handling is a common requirement at all application levels. Some freestanding implementations include support for many of the functions in the header **<string.h>**.

### Coding Guidelines

Issues of which headers must be provided by an implementation are outside the scope of coding guidelines. This is an application build configuration management issue.

---

implementation extensions

A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any strictly conforming program.[3)] 95

### Commentary

The C committee did not want to ban extensions. Common extensions were a source of material for both C90 and C99 documents. But the Committee does insist that any extensions do not alter the behavior of other constructs it defines. Extensions that do not change the behavior of any strictly conforming program are sometimes called *pure extensions*.

An implementation may provide additional library functions. It is a moot point whether they are actual extensions, since it is not suggested that libraries supplied by third parties have this status. The case for calling them extensions is particularly weak if the functionality they provide could have been implemented by the developer, using the same implementation but without those functions. However, there is an established practice of calling anything provided by the implementation that is not part of the standard an extension.

### Common Implementations

One of the most common extensions is support for inline assembler code. This is sometimes implemented by making the assembler code look like a function call, the name of the function being `asm`, e.g., `asm("ld r1, r2");`.

In the Microsoft/Intel world, the identifiers **`NEAR`**, **`FAR`**, and **`HUGE`** are commonly used as pointer type modifiers.

Implementations targeted at embedded systems (i.e., freestanding environments) sometimes use the `^` operator to select a bit from an object of a specified type. This is an example of a nonpure extension.

### Coding Guidelines

These days vendors do not try to tie customers into their products by doing things different from what the C Standard specifies. Rather, they include additional functionality; providing extensions to the language that many developers find useful. Source code containing many uses of a particular vendor's extensions is likely to be more costly to port to a different vendor's implementation than source code that does not contain these constructs.

Many developers accumulated most of their experience using a single implementation; this leads them into the trap of thinking that what their implementation does is what is supported by the standard. They may not be aware of using an extension. Using an extension through ignorance is poor practice.

Use of extensions is not in itself poor practice; it depends on why the extension is being used. An extension providing functionality that is not available through any other convenient means can be very attractive. Use of a construct, an extension or otherwise, after considering all other possibilities is good engineering practice.

A commonly experienced problem with vendor extensions is that they are not fully specified in the associated documentation. Every construct in the C Standard has been looked at by many vendors and its consequences can be claimed to have been very well thought through. The same can rarely be said to apply to a vendor's extensions. In many cases the only way to find out how an extension behaves, in a given situation, is to write test cases.

Some extensions interact with constructs already defined in the C Standard. For instance, some implementations[1] define a type, using the identifier **`bit`** to indicate a 1-bit representation, or using the punctuator `^` as a binary operator that extracts the value of a bit from its left operand (whose position is indicated by the right operand).[6] This can be a source of confusion for readers of the source code who have usually not been trained to expect this usage.

Experience shows that a common problem with the use of extensions is that it is not possible to quantify the amount of usage in source code. If use is made of extensions, providing some form of documentation for

the usage can be a useful aid in estimating the cost of future ports to new platforms.

Rev 95.1 The cost/benefit of any extensions that are used shall be evaluated and documented.

Dev 95.1 Use is made of extensions and:

- their use has been isolated within a small number of functions, or translation units,
- all functions containing an occurrence of an extension contain a comment at the head of the function definition listing the extensions used,
- test cases have to be written to verify that the extension operates as specified in the vendor's documentation. Test cases shall also be written to verify that use of the extension outside of the context in which it is defined is flagged by the implementation.

Some of the functions in the C library have the same name as functions defined by POSIX. POSIX, being an API-based standard (essentially a complete operating system) vendors have shown more interest in implementing the POSIX functionality.

**Example**

The following is an example of an extension, provided the VENDOR_X implementation is being used and the call to `f` is followed by a call to a trigonometric function, that affects the behavior of a strictly conforming program.

```
1  #include <math.h>
2
3  #if defined(VENDOR_X)
4  #include "vmath.h"
5  #endif
6
7  void f(void)
8  {
9  /*
10  * The following function call causes all subsequent calls
11  * to functions defined in <math.h> to treat their argument
12  *  values as denoting degrees, not radians.
13  */
14 #if defined(VENDOR_X)
15 switch_trig_to_degrees();
16 #endif
17 }
```

The following examples are pure extensions. Where might the coding guideline comments be placed?

```
1  /*
2   * This function contains assembler.
3   */
4  void f(void)
5  /*
6   * This function contains assembler.
7   */
8  {
9  /*
10  * This function contains assembler.
11  */
12 asm("make the, coffee"); /* How do we know this is an extension? */
13 } /* At least we can agree this is the end of the function. */
14
15 void no_special_comment(void)
16 {
```

```
17  asm("open the, biscuits");
18  }
19
20
21  void what_syntax_error(void)
22  {
23  asm wash up, afterwards
24  }
25
26  void not_isolated(void)
27  {
28  /*
29   * Enough standard C code to mean the following is not isolated.
30   */
31  asm wait for, lunch
32  }
```

footnote 2

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. 96

### Commentary

feature test macro

The definition of a macro, or lack of one, can be used to indicate the availability of certain functionality. The **#ifdef** directive providing a natural, language, based mechanism for checking whether an implementation supports a particular optional construct. The POSIX standard[4] calls macros, used to check for the availability (i.e., an implementations' support) of an optional construct, *feature test* macros.

### 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).

IEC 60559

### Other Languages

There is a philosophy of language standardization that says there should only be one language defined by a standard (i.e., no optional constructs). The Pascal and C90 Standard committees took this approach. Other language committees explicitly specify a multilevel standard; for instance, Cobol and SQL both define three levels of conformance.

C (and C++) are the only commonly used languages that contain a preprocessor, so this type of optional construct-handling functionality is not available in most other languages.

### Common Implementations

If an implementation does not support an optional construct appearing in source code, a translator often fails to translate it. This failure invariably occurs because identifiers are not defined. In the case of optional functions, which a translator running in a C90 mode to support implicit function declarations may not diagnose, there will be a link-time failure.

### Coding Guidelines

Use of a feature test macro highlights the fact that support for a construct is optional. The extent to which this information is likely to be already known to the reader of the source will depend on the extent to which a program makes use of the optional constructs. For instance, repeated tests of the `__STDC_IEC_559__` macro in the source code of a program that extensively manipulates IEC 60559 format floating-point values complicates the visible source and conveys little information. However, testing this macro in a small number

__STDC_IEC_559__ macro

of places in the source of a program that has a few dependencies on the IEC 60559 format is likely to provide useful information to readers.

Use of a feature test macro does not guarantee that a program correctly performs the intended operations; it simply provides a visual reminder of the status of a construct. Whether an **#else** arm should always be provided (either to handle the case when the construct is not available, or to cause a diagnostic to be generated during translation) is a program design issue.

### Example

```
#include <fenv.h>

void f(void)
{
#ifdef __STDC_IEC_559__
   fesetround(FE_UPWARD);
#endif /* The case of macro not being defined is ignored. */

#ifdef __STDC_IEC_559__
   fesetround(FE_UPWARD);
#else
   #error Support for IEC 60559 is required
#endif

#ifdef __STDC_IEC_559__
   fesetround(FE_UPWARD);
#else
   /*
    * An else arm that does nothing.
    * Does this count as handling the alternative?
    */
#endif
}
```

---

97 For example: example __STDC_IEC_559_

```
#ifdef __STDC_IEC_559__ /* FE_UPWARD defined */
   /* ... */
   fesetround(FE_UPWARD);
   /* ... */
#endif
```

---

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

### Commentary

If an implementation did reserve such an identifier, then its declaration could clash with one appearing in a strictly conforming program (probably leading to a diagnostic message being generated). The issue of reserved identifiers is discussed in more detail in the library section.

### 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.

### Common Implementations

In practice most implementation's system headers do define (and therefore could be said to reserve) identifiers whose spelling is not explicitly reserved for implementation use (see Table **??**). Many implementations that define additional keywords are careful to use the double underscore, `__`, prefix on their spelling. Such an identifier spelling is not always seen as being as readable as one without the double underscore. A commonly adopted renaming technique is to use a predefined macro name that maps to the double underscore name. The developer can always **`#undef`** this macro if its name clashes with identifiers declared in the source.

It is very common for an implementation to predefine several macros. These macros are either defined within the program image of the translator, or come into existence whenever one of the standard-defined headers is included. The names of the macros usually denote properties of the implementation, such as `SYSTYPE_BSD`, `WIN32`, `unix`, `hp9000s800`, and so on.

Identifiers defined by an implementation are visible via headers, which need to be included, and via libraries linked in during the final phase of translation. Most linkers have an *only extract the symbols needed* mode of working, which enables the same identifier name to be externally visible in the developers' translation unit and an implementation's library. The developers' translation unit is linked first, resolving any references to its symbol before the implementation's library is linked.

### Coding Guidelines

Coding guidelines cannot mandate what vendors (translator, third-party library, or systems integrator) put in the system headers they distribute. Coding guideline documents need to accept the fact that almost no commercial implementations meet this requirement.

Requiring that all identifiers declared in a program first be **`#undef`**'ed, on the basis that they may also be declared in a system header, would be overkill (and would only remove previously defined macro names). Most developers use a suck-it-and-see approach, changing the names of any identifiers that do clash.

Identifier name clashes between included header contents and developer written file scope declarations are likely to result in a diagnostic being issued during translation. Name usage clashes between header contents and block scope identifier definitions may sometimes result in a diagnostic; for instance, the macro replacement of an identifier in a block scope definition resulting in a syntax or constraint violation.

Measurements of code show (see Table 98.1) that most existing code often contains many declarations of identifiers whose spellings are reserved for use by implementations. Vendors are aware of this usage and often link against the translated output of developer written code before finally linking against implementation libraries (on the basis that resolving name clashes in favour of developer defined identifiers is more likely to produce the intended behavior).

Whether the cost of removing so many identifier spellings potentially having informative semantics, to readers of the source, associated with them is less than the benefit of avoiding possible name clash problems with implementation provided libraries is not known. No guideline recommendation is given here.

**Table 98.1:** Number of developer declared identifiers (the contents of any header was only counted once) whose spelling (the notation *[a-z]* denotes a regular expression, i.e., a character between *a* and *z*) is reserved for use by the implementation or future revisions of the C Standard. Based on the translated form of this book's benchmark programs.

| Reserved spelling | Occurrences |
|---|---|
| Identifier, starting with __, declared to have any form | 3,071 |
| Identifier, starting with *_[A-Z]*, declared to have any form | 10,255 |
| Identifier, starting with *wcs[a-z]*, declared to have any form | 1 |
| Identifier, with external linkage, defined in C99 | 12 |
| File scope identifier or tag | 6,832 |
| File scope identifier | 2 |
| Macro name reserved when appropriate header is `#included` | 6 |
| Possible macro covered identifier | 144 |
| Macro name starting with *E[A-Z]* | 339 |
| Macro name starting with *SIG[A-Z]* | 2 |
| Identifier, starting with *is[a-z]*, with external linkage (possibly macro covered) | 47 |
| Identifier, starting with *mem[a-z]*, with external linkage (possibly macro covered) | 108 |
| Identifier, starting with *str[a-z]*, with external linkage (possibly macro covered) | 904 |
| Identifier, starting with *to[a-z]*, with external linkage (possibly macro covered) | 338 |
| Identifier, starting with *is[a-z]*, with external linkage | 33 |
| Identifier, starting with *mem[a-z]*, with external linkage | 7 |
| Identifier, starting with *str[a-z]*, with external linkage | 28 |
| Identifier, starting with *to[a-z]*, with external linkage | 62 |

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

conforming program

### Commentary

Does the conforming implementation that accepts a particular program have to exist? Probably not. When discussing conformance issues, it is a useful simplification to deal with possible implementations, not having to worry if they actually exist. Locating an actual implementation that exhibits the desired behavior adds nothing to a discussion on conformance, but the existence of actual implementations can be a useful indicator for quality-of-implementation issues and the likelihood of certain constructions being used in real programs (the majority of real programs being translated by an extant implementation at some point).

### 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*:

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

1.3.14 well-formed program

### Coding Guidelines

Just because a program is translated without any diagnostics being issued does not mean that another translator, or even the same translator with a different set of options enabled, will behave the same way. A conforming program is acceptable to a conforming implementation. A strictly conforming program is acceptable to all conforming implementations.

90 strictly conforming program
use features of language/library

The cost of migrating a program from one implementation to all implementations may not be worth the benefits. In practice there is a lot of similarity between implementations targeting similar environments (e.g., the desktop, DSP, embedded controllers, supercomputers, etc.). Aiming to write software that will run within one of these specific environments is a much smaller task and can produce benefits at an acceptable cost.

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

implementation document

### Commentary

The formal validation process carried out by BSI (in the UK) and NIST (in the USA), when they were in the language-implementation validation business, checked that the implementation-defined behavior was documented. However, neither organization checked the accuracy of the documented behavior.

locale-specific behavior

### C90

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.

### Common Implementations

Many vendors include an appendix in their documentation where all implementation-defined behavior is collected together.

Of necessity a vendor will need to document extensions if their customers are to make use of them. Whether they document all extensions is another matter. One method of phasing out a superseded extension is to cease documenting it, but to continue to support it in the implementation. This enables programs that use the extension to continue being translated, but developers new to that implementation will be unlikely to make use of the extension (not having any documentation describing it).

### Coding Guidelines

For those cases where use of implementation-defined behavior is being considered, the vendor implementation-provided document will obviously need to be read. The commercially available compiler validation suites do not check implementation-defined behavior. It is recommended that small test programs be written to verify that an implementation's behavior is as documented.

---

**Forward references:** conditional inclusion (6.10.1), error directive (6.10.5), characteristics of floating types **<float.h>** (7.7), alternative spellings **<iso646.h>** (7.9), sizes of integer types **<limits.h>** (7.10), variable arguments **<stdarg.h>** (7.15), boolean type and values **<stdbool.h>** (7.16), common definitions **<stddef.h>** (7.17), integer types **<stdint.h>** (7.18). 101

---

footnote 4

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

strictly conforming program 90

use features of language/library

### Commentary

A strictly conforming program is acceptable to all conforming implementations.

### 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.

### Example

It is possible for a strictly conforming program to produce different output with different implementations, or even every time it is compiled:

```
1  #include <limits.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6  printf("INT_MAX=%d\n", INT_MAX);
7  printf("Translated date is %s\n", __DATE__);
8  }
```

---

conforming programs may depend on

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

### Commentary

What might such *nonportable* features be? The standard does not specify any construct as being nonportable. The only other instance of this term occurs in the definition of undefined behavior. One commonly used meaning of the term *nonportable* is a construct that is not likely to be available in another vendor's implementation. For instance, support for some form of inline assembler code is available in many implementations. Use of such a construct might not be considered as a significant portability issue.

undefined behavior

### C++

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

### Coding Guidelines

There are a wide range of constructs and environment assumptions that a program can make to render it nonportable. Many nonportable constructs tend to fall into the category of undefined and implementation-defined behaviors. Avoiding these could be viewed, in some cases, as being the same as avoiding nonportable constructs.

### Example

Relying on `INT_MAX` being larger than 32,767 is a dependence on a nonportable feature of a conforming implementation.

```
1  #include <limits.h>
2
3  _Bool f(void)
4  {
5  return (32767 < INT_MAX);
6  }
```

# References


1. Altrium BV. *C166/ST10 v8.0 C Cross-Compiler User's Guide*. Altrium BV, 2003.
2. Z. J. Ciechanowicz and A. C. D. Weever. The 'completeness' of the Pascal test suite. *Software–Practice and Experience*, 14(5):463–471, 1984.
3. IEEE. *IEEE 1003.3 Standard for Information Technology —Test Methods for Measuring Conformance to POSIX.1*. IEEE, 1991.
4. ISO. *ISO/IEC 9945-1:1990 Information technology —Portable Operating System Interface (POSIX)*. ISO, 1990.
5. D. M. Jones. Who guards the guardians? `www.knosof.co.uk/whoguard.html`, 1992.
6. Keil. *C Compiler manual*. Keil Software, Inc, ??? edition, May 2005.
7. X. Leroy. Formal certification of a compiler back-end *or*: Programming a compiler with a proof assistant. In *Proceedings of the 2006 POPL Conference*, pages 42–54, Jan. 2006.
8. G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–95, 2000.
9. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.
10. B. A. Wichmann and M. Davies. Experience with a compiler testing tool. Technical Report NPL Report DITC 138/89, National Physical Laboratory, England, 1989.
11. W. D. Young. *A verified code generator for a subset of Gypsy*. PhD thesis, The University of Texas at Austin, Dec. 1988.