

# **The New C Standard** (Excerpted material)

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

# 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.<sup>1)</sup>

1

## Commentary

The C Standard describes the behavior of programs (not always in complete detail, an implementation is given various amounts of leeway in translating some constructs). The behavior of implementations has to be deduced from the need to implement the described behavior of programs.

The committee took the view that programs are more important than implementations. This principle was and is used during the decision-making process of the C Standard Committee. Implementors sometimes argued that what their implementation did was/is important. The particular characteristics of an implementation can influence the usage of C language in programs, as can the characteristics of the host (e.g., the width of integer types supported). The Committee preferred to consider the extent of usage in existing programs and only became involved in the characteristics of implementations when there was widespread usage of a particular construct.

Rationale Existing code is important, existing implementations are not.

## 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.

## Other Languages

This exact wording appears in both the Cobol and Fortran standards (except the language name is changed and Fortran programs are “expressed” rather than “written”). Some language definitions do not explicitly specify whether they apply to programs or implementations. Pascal defines conformance requirements for both implementations and programs.

Gosling<sup>[4]</sup> *We intend that the behavior of every language construct is specified here, so that all implementations of Java will accept the same programs.*

## Common Implementations

base document

The C language was first described in 1975 in a Bell Labs technical report<sup>[7]</sup> (the successor to a language called B<sup>[7]</sup>). The more commonly known book *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie<sup>[9]</sup> was published in 1978. There was also a report published in the same year listing recent changes.<sup>[10]</sup> A second edition of this book was published after the ANSI C Standard was ratified<sup>[11]</sup> which updated its description of the language to follow that given in the newly published standard. There has been no republication since the C99 revision of the Standard.

The first edition of the Kernighan and Ritchie book describes what became known as *K&R C*. A large number of implementations were based on the original K&R book, few of them adhering exactly to the

specification it contained, (because it was open to interpretation). The term K&R compiler is often applied generically to translators that do not support function prototypes (an easily spotted characteristic).

As time has passed the number of implementations, in use, based on K&R C has dropped dramatically. But there is still source code in use that was written to the K&R specification. Vendors like to keep their customers happy by translating their existing code and many have added *support K&R* options to their products.

The base document for the library clause was the *1984 /usr/group Standard* published by the /usr/group Standard's Committee, Santa Clara, California, USA.

### Coding Guidelines

Coding guidelines invariably specify that ISO 9899 is the definitive document. The problem at the time of writing, and for the next few years, is that most implementations in common use follow the 1990 document, not the 1999 revision. Most of the changes involve additional functionality with very few changes to existing behavior. So most of the updates that are needed to move to C99 can be handled as new material.

Up until the mid 1990s portability considerations meant having to keep an eye on maintaining a K&R compatibility option. These days platforms that only support K&R are limited to a few niches, where there is insufficient market interest to make it worthwhile to create an ISO-conforming implementation.

Many students who are taught C++ are told that it is a superset of C. This was not always the case in C90 and is not true in C99 (where there is additional support for functionality not available in C++). Some compiler vendors offer a *C compiler* switch on their C++ compiler. Such switches do not always have the effect of creating a conforming C compiler. The issues of C/C++ compatibility are dealt with in the C++ subsections for each sentence.

## 2 It specifies

### Commentary

This list is not exhaustive in that permission is explicitly given for an implementation to have extensions.

imple-  
mentation  
extensions

### C++

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

### Coding Guidelines

These coding guideline subsections sometimes specify recommendations to be followed by developers for the usage of C language constructs.

## 3 — the representation of C programs;

### Commentary

The representation described is essentially the same as written text, with special meaning attributed to certain characters, singly or in sequences (e.g., end-of-line indicators). The representation also includes how the components of a C program are organized. In most cases files are used.

transla-  
tion phase  
1  
source files

The representation of C programs occurs at many different levels. There is the representation as it appears to the developer, the bytes read from media by the operating system, and the pattern of bits held on storage media. The representation we are interested in is the one that appears, to a developer, when viewed with an editor that supports the characters required by the C Standard.

### Common Implementations

The C language was first implemented on hosts that used the Ascii character set. The EBCDIC (Extended Binary-Coded-Decimal Interchange Code) character set is commonly used on mainframes and C can be represented using this character set.

EBCDIC

Use of C for embedded applications and the prominence of Japanese in this area meant that C was the first standardized language to allow the writing of programs that contained other character sets.

— the syntax and constraints of the C language;

**Commentary**

These two sets of specifications are set in concrete and must be implemented, as written, by every conforming implementation. These specifications appear in the Standard within clauses headed by "Syntax", or by "Constraints".

**C++**

conformance  
constraint

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.

diagnostic  
shall produce

**Common Implementations**

Some implementations add additional syntax. Adding additional constraints, or relaxing the existing ones is not commonly seen in implementations, but it does occur.

**Coding Guidelines**

Constructs that violate C syntax or constraints are required to be diagnosed by a conforming implementation. Working programs rarely contain such constructs (unless there is a bug in the implementation; for instance, gcc allows semicolons to be omitted in several places). Duplicating these requirements in coding guidelines does not add value.

Implementations that provide extensions to standard C do not always fully define these extensions. In particular they often define how an extension may be used but fail to define what the constraints on its use are. The coding guideline issues relating to use of extensions is discussed elsewhere.

extensions ??  
cost/benefit

— the semantic rules for interpreting C programs;

**Commentary**

These rules appear under the clause heading *Semantics*, or sometimes *Description*, along with definitions of conformance. The behavior arising out of these semantic rules is what developers use to write programs that have an external effect.

The standard talks about an abstract machine and how programs are to be interpreted as-if running under it. Unfortunately, the specifications given in the standard are not worded in a form that directly addresses the properties of this machine; as such, this machine is never fully defined. However, the semantic rules specified in the C Standard are not all set in stone. An implementation may be required to select among several alternatives (these form the category of unspecified behaviors), chose its own behavior (these form the category of implementation-defined behaviors), or the standard may not impose any requirements on the behavior (these form the category of undefined behaviors).

**C++**

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

**Coding Guidelines**

A good first approximation to a set of coding guidelines is to recommend against the use of constructs whose semantic rules can vary across implementations (Annex J summarizes these and the majority of the rules in the MISRA C guidelines are based on this principle). While many of these Coding guideline subsections discuss the effect of implementation differences, these are only treated as a possible contributing factor to the primary consideration (i.e., cost) and not as a rationale in their own right.

conformance  
behavior

MISRA  
guideline  
recommendations  
selecting

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

### Commentary

The C language had its beginnings in solving practical problems. Ignoring the representational issues of data was not a viable option. The Committee did adopt a specification for a set of I/O concepts (e.g., streams, binary and text files) and functions for manipulating them into the C library (the base document provided the underlying models). <sup>1</sup> base document

The underlying unit of input is the byte. The standard does not require that any sequence of bits within any byte have a particular interpretation (the functions provided by the `<ctype.h>` header can be applied to them, just like any other numeric quantity).

### Other Languages

Some language standards committees have taken the view that I/O was not an important aspect of the language and provided a minimal set of functionality in this area, saying nothing about representational issues. In other languages, for instance Cobol, I/O is a significant part of that languages' specification.

### C++

The C++ Standard is silent on this issue.

### Common Implementations

A form of input not explicitly dealt with in the standard is the reading/writing data from/to registers, or I/O ports. This is a common form of I/O in freestanding environments.

Many implementations that support such functionality use the **volatile** type qualifier or some extension that allows objects to be placed at known locations in storage; reading values from such objects cause input to take place. Here the representation of the input value is interpreted according to the type of object through which it is accessed. <sup>type qualifier syntax</sup>

### Coding Guidelines

The issues involved in converting this input data into some internal form is an application domain issue that is outside the scope of these coding guidelines. For instance, a floating-point number presented as a sequence of characters, on an input stream, may contain more accuracy than can be represented by the host. There is a guideline recommendation dealing with the use of representation information. <sup>?? representation information using</sup>

7— the representation of output data produced by C programs;

### Commentary

The standard does not specify any representation in terms of bit patterns, or pixels on a display device. However, the standard does specify some ordering requirements on output data. Data written out can be read back in to produce the same value. Output written to a display device will appear in the order it is written (but nothing is said about left-to-right, right-to-left, top-to-bottom, or any other visible ordering on the device). <sup>writing direction locale-specific</sup>

C supports two forms of representation on output, text and binary. Text I/O is structured into lines of characters (there can be a great deal of variability in the external representation of characters written to text streams), while binary I/O is an ordered sequence of bytes.

### Other Languages

In some application domains organizing the output data is a substantial part of the problem. Some languages, for instance Cobol, have mechanisms that provide application domain related control (in the case of Cobol the formatting of numeric quantities) of the output produced by a program.

### Common Implementations

In most cases, implementations use the same representation for output data as they use for input data.

### Coding Guidelines

The coding guideline subsections only discuss the representation of output data to the extent that it may be used as input data to other programs written in C.

limits specify

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

8

**Commentary**

The Committee recognized that all implementations impose some limits on the size of programs that can be translated. They decided to face up to this issue by specifying minimum requirements. By specifying a list of limits, the Committee is attempting to guarantee a minimal level of support, for programs, by all conforming implementations. The limits were seen as a floor that implementations should strive to exceed, not as a ceiling they could stop at.

Since the C90 Standard was written, the average amount of memory available to translators, on the majority of hosts, has increased significantly. For C99 the nominal translator host memory limit was increased to 512 K.

**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.

**Common Implementations**

Few implementations document all the limits they impose. This is usually because of the use of dynamic data structures, which means that their only fixed limit is the amount of memory available during translation.

---

This International Standard does not specify

9

**Commentary**

The C committee is up front about what the C Standard is not about.

**C++**

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

**Other Languages**

Some languages standards include a list of items not specified by their respective documents, while others do not. Many of the items listed in the C Standard also appear in the Fortran Standard.

**Coding Guidelines**

A set of coding guideline recommendations cannot hope to cover every issue that occurs in source code. Delimiting the areas not covered by a set of guidelines is as important as specifying those areas covered.

---

— the mechanism by which C programs are transformed for use by a data-processing system;

10

**Commentary**

The standard uses the term *translator* to disassociate itself from known implementation techniques for transforming programs, such as compilers and interpreters. There is no requirement that the transformation process use programs that have been written in C, although the library does contain many of the support functions needed in the implementation of such a translator.

All suggestions requiring some mechanism to exist for passing options to a translator, at translation-time, were turned down by the Committee.

environ-  
mental  
limits  
translation  
limits

coding  
guidelines  
background to

program  
transformation  
mechanism

Rationale One proposal long entertained by the C89 Committee was to mandate that each implementation have a translation-time switch for turning off extensions and making a pure Standard-conforming implementation. It was pointed out, however, that virtually every translation-time switch setting effectively creates a different “implementation”, however close may be the effect of translating with two different switch settings. Whether

an implementor chooses to offer a family of conforming implementations, or to offer an assortment of non-conforming implementations along with one that conforms, was not the business of the C89 Committee to mandate. The Standard therefore confines itself to describing conformance, and merely suggests areas where extensions will not compromise conformance.

Many existing translators operate in a single pass and continued support for this form of implementation a consideration for many members of WG14 when considering the specification of C syntax and semantics.

implementation  
single pass

## Other Languages

Some standards use the term *language processor* to define what C calls a translator.

## Common Implementations

The most common transformation mechanism (implementation technique) is to compile the program's source code to some form of machine code. This machine code could represent the instructions of a real processor (in the sense of being able to hold it in one's hand), or some virtual machine whose operations are performed in software (usually called an interpreter). In a few cases processors have been designed to execute a representation of some language directly. The Bell Labs CRISP processor<sup>[2]</sup> was designed to efficiently support the execution of translated C programs (although the extent to which its instruction set might be claimed to be *C-like* is open to debate); the Symbolics 3600 processor used Lisp as its machine language;<sup>[3]</sup> the Novix NC4016<sup>[12]</sup> used Forth as its instruction set.

The ability to execute source code on a line-by-line basis is rarely provided (the Extensible Interactive C system<sup>[1]</sup> is an exception). In such an approach the standard still requires (if a vendor wanted to claim that their product was a conforming implementation) that the entire program's source code, even the unexecuted portions, be analyzed for syntax and constraint violations. As well as providing an interactive mode, the Extensible Interactive C system also provides a batch mode.

An advantage of the virtual machine approach is that the generated code can be executed unchanged on a wide variety of different processors, given the availability of a software interpreter. This is how Java achieves its portability. Another advantage of this approach is the compactness of the generated code. In applications where code size is more important than performance, it can be the deciding factor in choosing an interpretive approach.

The performance advantage obtained from using a cache shows how the execution time characteristics of many applications is to repeatedly execute the same sequences of instructions within short time periods. It is possible to make use of this execution time characteristic to have the best of both worlds— execution performance and compact code. The less-frequently executed portions of a program exist in virtual machine code form and the frequently executed portions in host processor machine code. For a VLIW processor, Hoogerbrugge<sup>[6]</sup> was able to obtain a 50% reduction in code size for a negligible increase in execution time.

cache

Whatever the mechanism used to transform C programs, it will invariably support some form of `-I` and `-D` options. These two options are almost universally used by C translators for specifying search paths for `#include` files and for defining macros (on the command line), respectively.

`#include`  
places to search  
for  
macro  
object-like

Some translators access environment variables, of their host operating, to obtain the values of attribute that vary between different hosts. For instance, search paths for the `#include` directive, or the number of processors available to a program (when generating parallelization code<sup>[18]</sup>).

`#include`  
places to search  
for

Mixed forms of translation are being researched. Here translation of selected portions of a program occurs during the execution of that program. The advantage of this dynamic compilation approach is that it is possible to make use of runtime information to specialize the code, enhancing performance (provided the specialized execution savings are greater than the overhead of a dynamic compilation). The DyC toolset<sup>[5, 13]</sup> has achieved some interesting results.

Many so-called *number crunching* applications are written in Fortran. A variety of parallel and vector processors have been built to reduce the execution time of such programs, which has entailed producing translators capable of vectorizing and parallelizing Fortran. One way to tap into this existing technology is to translate C source to Fortran.<sup>[8]</sup>

Downloading programs onto mobile devices, where they might only be executed once, is becoming more common. In this environment, the consumption of electrical power is an important consideration. A study by Palm and Moss<sup>[16]</sup> performed a cost/benefit analysis of translating code on the client or server, with or without optimization. The energy quantities considered were:  $E_{download}$ , energy consumed by the wireless card while downloading code;  $E_{wait-download}$ , energy consumed by the client while waiting for code to download;  $E_{wait-compile}$ , energy consumed while waiting for code to compile or optimize on the server;  $E_{compile}$ , energy for compiling or optimizing on the client; and  $E_{run}$ , energy for running the compiled application on the client.

**Coding Guidelines**

Providing different options to a translator effectively creates different translators. The purpose of specifying options is to change the behavior of a translator, otherwise there is no point in specifying it. Whether use of an option radically changes the behavior of a translator (e.g., by enabling language extensions, changing the alignment of objects in memory, or selecting different hosts as the execution environment) or has no noticeable effect on the external output of the generated program image (e.g., it changes the format of the listing file, or causes debug information to be generated), is outside the scope of these coding guidelines. Selecting translation-time options is part of the configuration management for a project.

Your author has never seen a set of coding guidelines that apply to make-files (apart from layout conventions) and would suggest that work on such a set is long overdue.

---

— the mechanism by which C programs are invoked for use by a data-processing system; 11

**Commentary**

The standard specifies the behaviors of C constructs. Specifying mechanisms for executing the generated program images serves no useful purpose at the level of abstraction at which the standard operates. Whichever mechanism is used, for a hosted implementation, the standard requires that a function called `main` be called by the execution environment.

program startup

**Common Implementations**

The output generated by a translator is usually written to a file. To indicate that this file can be executed, as a program, it might be given the extension `.exe` (under Microsoft Windows), or have its execute-bit set (under a POSIX-compliant operating systems such as Linux). Existing practices for invoking a program include giving the program name on the command line, clicking on icons, and having the program automatically executed on computer startup.

In a freestanding environment the program image may be stored in read-only memory at a location that the host processor jumps to when it is initialized (which usually happens by default when power is first applied). The act of switching on, or resetting, is the mechanism for invoking the program.

freestanding environment startup

---

— the mechanism by which input data are transformed for use by a C program; 12

**Commentary**

The standard is not concerned with how data is represented on media (which may be held on a hard disk, paper tape, or any other media) or an interactive device. It is the implementation’s responsibility to map data from the bits held on a storage device to the input values they represent to a C program.

**Coding Guidelines**

Programs that want to access devices at a level below that specified by the C Standard are outside the scope of these coding guidelines.

**Example**

Specifying that a file is to be opened in text mode will cause the input to be treated as a series of lines. How lines are represented by the host file system is outside the scope of the C Standard.

input data mechanism transformed

end-of-line representation

13— the mechanism by which output data are transformed after being produced by a C program;

### Commentary

The Committee recognized that a program image may be executing within an encompassing environment. How this environment transforms data after it has been output by a program is outside the scope of the C Standard. The standard specifies an intended external effect for operations that perform output. It does not specify a *final resting place* for this external effect. It may be characters appearing on a display device, or bits being written to a storage device, or many other possibilities.

### Common Implementations

At the host environment level (operating system) the sequences of bytes output by a C program are rarely modified until they reach the lower-levels of device drivers. Bytes sent over serial links may have parity bits added to them, blocks of bytes written to media may include file system information, and so on.

### Coding Guidelines

Programs that are concerned with how output data are transformed once it has been generated by a program image are outside the scope of these coding guidelines.

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

footnote  
1

### Commentary

The Rationale puts the case very well:

C code can be portable. Although the C language was originally born with the UNIX operating system on the DEC PDP-11, it has since been implemented on a wide variety of computers and operating systems. It has also seen considerable use in cross-compilation of code for embedded systems to be executed in a free-standing environment. The C89 Committee attempted to specify the language and the library to be as widely implementable as possible, while recognizing that a system must meet certain minimum criteria to be considered a viable host or target for the language.

Rationale

C code can be non-portable. Although it strove to give programmers the opportunity to write truly portable programs, the C89 Committee did not want to force programmers into writing portably, to preclude the use of C as a “high-level assembler:” the ability to write machine-specific code is one of the strengths of C. It is this principle which largely motivates drawing the distinction between strictly conforming program and conforming program (§4).

Avoid “quiet changes.” Any change to widespread practice altering the meaning of existing code causes problems. Changes that cause code to be so ill-formed as to require diagnostic messages are at least easy to detect. As much as seemed possible consistent with its other goals, the C89 Committee avoided changes that quietly alter one valid program to another with different semantics, that cause a working program to work differently without notice. In important places where this principle is violated, both the C89 Rationale and this Rationale point out a QUIET CHANGE.

A standard is a treaty between implementor and programmer. Some numerical limits were added to the Standard to give both implementors and programmers a better understanding of what must be provided by an implementation, of what can be expected and depended on to exist. These limits were, and still are, presented as minimum maxima (that is, lower limits placed on the values of upper limits specified by an implementation) with the understanding that any implementor is at liberty to provide higher limits than the Standard mandates. Any program that takes advantage of these more tolerant limits is not strictly conforming, however, since other implementations are at liberty to enforce the mandated limits.

Keep the spirit of C. The C89 Committee kept as a major goal to preserve the traditional spirit of C. There are many facets of the spirit of C, but the essence is a community sentiment of the underlying principles on which the C language is based. Some of the facets of the spirit of C can be summarized in phrases like

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.
- Make it fast, even if it is not guaranteed to be portable.

The last proverb needs a little explanation. The potential for efficient code generation is one of the most important strengths of C. To help ensure that no code explosion occurs for what appears to be a very simple operation, many operations are defined to be how the target machine's hardware does it rather than by a general abstract rule. An example of this willingness to live with what the machine does can be seen in the rules that govern the widening of char objects for use in expressions: whether the values of char objects widen to signed or unsigned quantities typically depends on which byte operation is more efficient on the target machine.

One of the goals of the C89 Committee was to avoid interfering with the ability of translators to generate compact, efficient code. In several cases the C89 Committee introduced features to improve the possible efficiency of the generated code; for instance, floating point operations may be performed in single-precision if both operands are float rather than double.

At the WG14 meeting in Tokyo, Japan, in July 1994, the original principles were re-endorsed and the following new ones were added:

Support international programming. During the initial standardization process, support for internationalization<sup>[14]</sup> was something of an afterthought. Now that internationalization has become an important topic, it should have equal visibility. As a result, all revision proposals shall be reviewed with regard to their impact on internationalization as well as for other technical merit.

Codify existing practice to address evident deficiencies. Only those concepts that have some prior art should be accepted. (Prior art may come from implementations of languages other than C.) Unless some proposed new feature addresses an evident deficiency that is actually felt by more than a few C programmers, no new inventions should be entertained.

Minimize incompatibilities with C90 (ISO/IEC 9899:1990). It should be possible for existing C implementations to gradually migrate to future conformance, rather than requiring a replacement of the environment. It should also be possible for the vast majority of existing conforming programs to run unchanged.

Minimize incompatibilities with C++. The Committee recognizes the need for a clear and defensible plan for addressing the compatibility issue with C++. The Committee endorses the principle of maintaining the largest common subset clearly and from the outset. Such a principle should satisfy the requirement to maximize overlap of the languages while maintaining a distinction between them and allowing them to evolve separately.

The Committee is content to let C++ be the big and ambitious language. While some features of C++ may well be embraced, it is not the Committee's intention that C become C++.

Maintain conceptual simplicity. The Committee prefers an economy of concepts that do the job. Members should identify the issues and prescribe the minimal amount of machinery that will solve the problems. The Committee recognizes the importance of being able to describe and teach new concepts in a straight-forward and concise manner.

During the revision process, it was important to consider the following observations:

- Regarding the 11 principles, there is a trade-off between them—none is absolute. However, the more the Committee deviates from them, the more rationale will be needed to explain the deviation.
- There had been a very positive reception of the standard from both the user and vendor communities.
- The standard was not considered to be broken. Rather, the revision was needed to track emerging and/or changing technologies and internationalization requirements.
- Most users of C view it as a general-purpose high-level language. While higher-level constructs can be added, they should be done so only if they don't contradict the basic principles.
- There are a good number of useful suggestions to be found from the public comments and defect report processing.

Areas to which the Committee looked when revising the C Standard included:

- Incorporate AMD1.
- Incorporate all Technical Corrigenda and records of response.
- Current defect reports.
- Future directions in current standard.
- Features currently labeled obsolescent.
- Cross-language standards groups work.
- Requirements resulting from JTC 1/SC 2 (character sets).
- The evolution of C++.
- The evolution of other languages, particularly with regard to interlanguage communication issues.
- Other papers and proposals from member delegations, such as the numerical extensions Technical Report which was proposed by J11.
- Other comments from the public at large.
- Other prior art.

## C++

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

### Coding Guidelines

What are coding guidelines designed to promote?

[coding  
guidelines  
introduction](#)

Rev 14.1

The first paragraph on page one of a coding guidelines document shall state the purpose of those guidelines and the benefits expected to accrue from adherence to them.

15 It is intended for use by implementors and programmers.

### Commentary

One argument, used by the Committee, against the use of a formal definition language for specifying the requirements, in the standard, was that programmers would have difficulty understanding it. Given the small number of copies of the document actually sold by standards bodies, this seems to be a moot point. This situation may change with C99 thanks to a standards organization in the USA being willing to sell electronic copies at a reasonable price.

Although written using English prose, the wording of the standard is highly stylized. Readers need to become familiar with the conventions used if they are to correctly interpret its contents. The ISO directives also require that:

ISO Direc-  
tives, part 3

*To achieve this objective, the International Standard shall*

...

— *be comprehensible to qualified persons who have not participated in its preparation.*

treaty 14

During the initial development of the C Standard by the ANSI committee, the idea of the document being a *treaty* between implementor and developer was voiced by many members of that committee. The Rationale discusses this issue. Although many members of the ISO C committee also hold this view of a standard being a treaty, there are some members who view the document as being a specification.

### Coding Guidelines

coding  
guidelines  
introduction

Who are the intended audience of these coding guidelines?

They are intended to be read by managers wanting to select a set of guideline recommendations applicable to their business model, authors of local coding guideline documents and training materials, and vendors producing tools to enforce them. While some developers may chose to read these coding guidelines for educational purposes, there is no obvious cost/benefit justification for requiring all developers to systematically read them.

---

— the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor; 16

### Commentary

The standard does not require an implementation to fail to translate and execute a program that exceeds a size or complexity (how complexity might be measured is not specified) limit. Although such a requirement could increase program portability (they would have to be rewritten to reduce their size or complexity, making it more likely that they would be able to execute on a larger range of hosts). However, restricting those programs that may be translated for these reasons is counterproductive. Some translation limits effectively specify minimum bounds on program and data size that must be translated and executed by an implementation. However, they are lower, not upper limits.

limit  
external identifiers

### Common Implementations

It is not usually the size or complexity of programs that give translators problems. Optimizers like to keep all the information associated with a given function in memory while it is being translated. Functions that are large (complexity itself is rarely a problem) run the risk of having the translator run out of memory while generating machine code for them.

During execution all programs have a limit on the memory available to them to allocate for object storage. Most implementations will allocate storage until insufficient is available; problem execution usually fails in some way at this point.

### Coding Guidelines

Programs that are too large or complex to be translated are fail-safe in the sense that developer attention is needed to solve the problem. Ensuring that programs do not run out of storage during execution, or that their execution terminates within a given time frame, are issues that are outside the scope of these coding guidelines. Some coding guideline documents address the storage capacity issue by prohibiting the use of constructs that prevent a program's total storage requirements from being known at translation time.<sup>[15]</sup>

---

— all minimal requirements of a data-processing system that is capable of supporting a conforming implementation. 17

**Commentary**

This statement is not quite true. When deciding on values for the minimum translation limits, the C99 Committee had in mind a translation host with a total of 256 K of memory. The requirements needed to support a conforming implementation are considered to be a quality-of-implementation issue. Implementation vendors are left to respond to customer demands.

limit  
external identi-  
fiers

**Common Implementations**

Implementations have been created on hosts having a total memory size of 64 K.

## References

1. E. J. Breen. *Extensible Interactive C*. [eic.sourceforge.net](http://eic.sourceforge.net), June 2000.
2. D. R. Ditzel and A. D. Berenbaum. Design tradeoffs to support the C programming language in the CRISP microprocessor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems-ASPLOSII*, pages 158–163. IEEE Computer Society Press, Oct. 1987.
3. A. D. Flatau. A Pascal to Lisp translator for the Symbolics 3600 Lisp machine. Thesis (m.s.), University of Texas at Austin, Austin, TX, 1985.
4. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison–Wesley, 1996.
5. B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: An expressive annotation-directed dynamic compiler for C. Technical Report TR-97-03-03, University of Washington, Department of Computer Science and Engineering, Mar. 1997.
6. J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software–Practice and Experience*, 29(11):1005–1023, Sept. 1999.
7. S. C. Johnson and B. W. Kernighan. The programming language B. Technical Report 8, Bell Telephone Laboratories, Jan. 1973.
8. R. L. Kennell and R. Eigenmann. Automatic parallelization of C by means of language transcription. In S. Chatterjee, J. Prins, L. Carter, J. Ferrante, Z. Li, D. C. Sehr, and P.-C. Yew, editors, *Proceedings of the 11<sup>th</sup> International Workshop on Languages and Compilers for Parallel Computing (LCPC-98)*, Lecture Notes in Computer Science, pages 166–180. Springer, 1998.
9. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc, 1978.
10. B. W. Kernighan and D. M. Ritchie. Recent changes to C. from Ritchie’s web page, Nov. 1978.
11. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc, 1988.
12. P. J. Koopman Jr. *Stack Computers the new wave*. Mountain View Press, 1989.
13. M. U. Mock. *Automatic Selective Dynamic Compilation*. PhD thesis, University of Washington, 2002.
14. S. M. O’Donnell. *Programming for the World*. Prentice Hall, Inc, 1994.
15. U. M. of Defence. *Defence Standard 00-55. Requirements for safety related software in defence equipment. Part 2: Guidance*. UK Ministry of Defence, Aug. 1997.
16. J. Palm and J. E. B. Moss. When to use a compilation service? In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems, LCTES’02*, pages 194–203. ACM, June 2002.
17. D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. The C programming language. Technical Report 31, Bell Telephone Laboratories, Oct. 1975.
18. Sun. *C User’s Guide*. Sun Microsystems, Inc, Palo Alto, CA, USA, revision a edition, May 2000.