

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

---

**An Economic and Cultural Commentary**

**Derek M. Jones**

derek@knosof.co.uk

*keyword:* one of

<b>auto</b>	<b>enum</b>	<b>restrict</b>	<b>unsigned</b>
<b>break</b>	<b>extern</b>	<b>return</b>	<b>void</b>
<b>case</b>	<b>float</b>	<b>short</b>	<b>volatile</b>
<b>char</b>	<b>for</b>	<b>signed</b>	<b>while</b>
<b>const</b>	<b>goto</b>	<b>sizeof</b>	<b>_Bool</b>
<b>continue</b>	<b>if</b>	<b>static</b>	<b>_Complex</b>
<b>default</b>	<b>inline</b>	<b>struct</b>	<b>_Imaginary</b>
<b>do</b>	<b>int</b>	<b>switch</b>	
<b>double</b>	<b>long</b>	<b>typedef</b>	
<b>else</b>	<b>register</b>	<b>union</b>	

### Commentary

The keywords **const** and **volatile** were not in the base document. The identifier **entry** was reserved by the base document but the functionality suggested by its name (Fortran-style multiple entry points into a function) was never introduced into C.

The standard specifies, in a footnote, the form that any implementation-defined keywords should take.

### C90

Support for the keywords **restrict**, **\_Bool**, **\_Complex**, and **\_Imaginary** is new in C99.

### C++

The C++ Standard includes the additional keywords:

<b>bool</b>	<b>mutable</b>	<b>this</b>
<b>catch</b>	<b>namespace</b>	<b>throw</b>
<b>class</b>	<b>new</b>	<b>true</b>
<b>const_cast</b>	<b>operator</b>	<b>try</b>
<b>delete</b>	<b>private</b>	<b>typeid</b>
<b>dynamic_cast</b>	<b>protected</b>	<b>typename</b>
<b>explicit</b>	<b>public</b>	<b>using</b>
<b>export</b>	<b>reinterpret_cast</b>	<b>virtual</b>
<b>false</b>	<b>static_cast</b>	<b>wchar_t</b>
<b>friend</b>	<b>template</b>	

The C++ Standard does not include the keywords **restrict**, **\_Bool**, **\_Complex**, and **\_Imaginary**. However, identifiers beginning with an underscore followed by an uppercase letter is reserved for use by C++ implementations (17.4.3.1.2p1). So, three of these keywords are not available for use by developers.

In C the identifier `wchar_t` is a typedef name defined in a number of headers; it is not a keyword.

The C99 header `<stdbool.h>` defines macros named `bool`, `true`, `false`. This header is new in C99 and is not one of the ones listed in the C++ Standard as being supported by that language.

### Other Languages

Modula-2 requires that all keywords be in uppercase. In languages where case is not significant keywords can appear in a mixture of cases.

### Common Implementations

The most commonly seen keyword added by implementations, as an extension, is **asm**. The original K&R specification included **entry** as a keyword; it was reserved for future use.

The processors that tend to be used to host freestanding environments often have a variety of different memory models. Implementation support for these different memory models is often achieved through the use of additional keywords (e.g., **near**, **far**, **huge**, **segment**, and **interrupt**). The C for embedded systems TR defines the keywords **\_Accum**, **\_Fract**, and **\_Sat**.

base doc-  
ument

footnote  
28

Embed-  
ded C TR

## Coding Guidelines

One of the techniques used by implementations, for creating language extensions is to define a new keyword. If developers decided to deviate from the guideline recommendation dealing with the use of extensions, some degree of implementation vendor independence is often desired. Some method for reducing the impact of the use of these keywords, on a program's portability, is needed. The following are a number of techniques:

?? extensions  
cost/benefit

- *Use of macro names.* Here a macro name is defined and this name is used in place of the keyword (which is the macro's body). This works well when there is no additional syntax associated with the keyword and the semantics of a program are unchanged if it is not used. Examples of this type of keyword include **near**, **far** and **huge**.
- *Limiting use of the keyword in source code.* This is possible if the functionality provided by the keyword can be encapsulated in a function that can be called whenever it is required.
- *Conditional compilation.* Littering the source code with conditional compilation directives is really a sign of defeat; it has proven impossible to control the keyword usage.

If there are additional tokens associated with an extension keyword, there are advantages to keeping all of these tokens on the same line. It simplifies the job of stripping them from the source code. Also a number of static analysis tools have an option to ignore all tokens to the end of line when a particular keyword is encountered. (This enables them to parse source containing these syntactic extensions without knowing what the syntax might be.)

## Usage

Usage information on preprocessor directives is given elsewhere (see Table ??).

**Table 788.1:** Occurrence of keywords (as a percentage of all keywords in the respective suffixed file) and occurrence of those keywords as the first and last token on a line (as a percentage of occurrences of the respective keyword; for .c files only). Based on the visible form of the .c and .h files.

Keyword	.c Files	.h Files	% Start of Line	% End of Line	Keyword	.c Files	.h Files	% Start of Line	% End of Line
<b>if</b>	21.46	15.63	93.60	0.00	<b>const</b>	0.94	0.80	35.50	0.30
<b>int</b>	11.31	13.40	47.00	5.30	<b>switch</b>	0.75	0.77	99.40	0.00
<b>return</b>	10.18	12.23	94.50	0.10	<b>extern</b>	0.61	0.71	99.60	0.40
<b>struct</b>	8.10	10.33	38.90	0.30	<b>register</b>	0.59	0.64	95.00	0.00
<b>void</b>	6.24	10.27	28.70	18.20	<b>default</b>	0.54	0.58	99.90	0.00
<b>static</b>	6.04	8.07	99.80	0.60	<b>continue</b>	0.49	0.33	91.30	0.00
<b>char</b>	4.90	5.08	30.50	0.20	<b>short</b>	0.38	0.28	16.00	1.00
<b>case</b>	4.67	4.81	97.80	0.00	<b>enum</b>	0.20	0.27	73.70	1.80
<b>else</b>	4.62	3.30	70.20	42.20	<b>do</b>	0.20	0.25	87.30	21.30
<b>unsigned</b>	4.17	2.58	46.80	0.10	<b>volatile</b>	0.18	0.17	50.00	0.00
<b>break</b>	3.77	2.44	91.80	0.00	<b>float</b>	0.16	0.17	54.00	0.70
<b>sizeof</b>	2.23	2.24	11.30	0.00	<b>typedef</b>	0.15	0.09	99.80	0.00
<b>long</b>	2.23	1.49	10.10	1.70	<b>double</b>	0.14	0.08	53.60	3.10
<b>for</b>	2.22	1.06	99.70	0.00	<b>union</b>	0.04	0.06	63.30	6.20
<b>while</b>	1.23	0.95	85.20	0.10	<b>signed</b>	0.02	0.01	27.20	0.00
<b>goto</b>	1.23	0.89	94.10	0.00	<b>auto</b>	0.00	0.00	0.00	0.00

## Semantics

789 The above tokens (case sensitive) are reserved (in translation phases 7 and 8) for use as keywords, and shall not be used otherwise.

translation phase  
7

### Commentary

A translator converts all identifiers with the spelling of a keyword into a keyword token in translation phase 7. This prevents them from being used for any other purpose during or after that phase. Identifiers that have the spelling of a keyword may be defined as macros, however there is a requirement in the library section that such definitions not occur prior to the inclusion of any library header. These identifiers are deleted after translation phase 4.

translation phase  
4

In translation phase 8 it is possible for the name of an externally visible identifier, defined using another language, to have the same spelling as a C keyword. A C function, for instance, might call a Fortran subroutine called xyz. The function xyz in turn calls a Fortran subroutine called default. Such a usage does not require a diagnostic to be issued.

### Other Languages

Most modern languages also reserve identifiers with the spelling of keywords purely for use as keywords. In the past a variety of methods for distinguishing keywords from identifiers have been adopted by language designers, including:

- *By the context in which they occur (e.g., Fortran and PL/I).* In such languages it is possible to declare an identifier that has the spelling of a keyword and the translator has to deduce the intended interpretation from the context in which it occurs.
- *By typeface (e.g., Algol 68).* In such languages the developer has to specify, when entering the text of a program into an editor, which character sequences are keywords. (Conventions vary on which keys have to be pressed to specify this treatment.) Displays that only support a single font might show keywords in bold, or underline them.
- *Some other form of visually distinguishable feature (e.g., Algol 68, Simula).* This feature might be a character prefix (e.g., **'begin** or **.begin**), a change of case (e.g., keywords always written using uppercase letters), or a prefix and a suffix (e.g., **'begin'**).

The term *stropping* is sometimes applied to the process of distinguishing keywords from identifiers.

Lisp has no keywords, but lots of predefined functions.

In some languages (e.g., Ada, Pascal, and Visual Basic) the spelling of keywords is not case sensitive.

### Common Implementations

Linkers are rarely aware of C keywords. The names of library functions, translated from other languages, are unlikely to be an issue.

### Coding Guidelines

A library function that has the spelling of a C keyword is not callable directly from C. An interface function, using a different spelling, has to be created. C coding guidelines are unlikely to have any influence over other languages, so there is probably nothing useful that can be said on this subject.

---

The keyword `_Imaginary` is reserved for specifying imaginary types.<sup>59)</sup>

790

### Commentary

This sentence was added by the response to DR #207. The Committee felt that imaginary types were not consistently specified throughout the standard. The approach taken was one of minimal disturbance, modifying the small amount of existing wording, dealing with these types. Readers are referred to Annex G for the details.

footnote  
59

---

59) One possible specification for imaginary types appears in Annex G.

791

### Commentary

This footnote was added by the response to DR #207.

# References