The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.10.8 Predefined macro names

macro name predefined

The following macro names[151)] shall be defined by the implementation: 2004

Commentary

This is a requirement on the implementation. These macro names are commonly referred to as the *predefined macros*. There is no clause header (i.e., Description, Semantics, Constraints) given for the requirements in this clause.

Some of the macros specified in the following C sentences expand to string literals. These string literals do not have any special properties associated with them (e.g., there is no requirement that any of them share the same storage).

string literal distinct array

Other Languages

Many languages (e.g., Ada and Fortran) support a large number of predefined identifiers (which may be intrinsic or part of a predefined library) supplying a variety of information.

Common Implementations

Most implementations also define macros, internally within the translator (i.e., not in headers), that provide information on the identity of the translator (e.g., a representation of the vendor name) and the host environment (particularly those vendors that supply implementations on a number of environment and host processors).

Table 2004.1: Occurrence of predefined macro names (as a percentage of all predefined macro names; a total of 1,826). Based on the visible form of the `.c` and `.h` files.

Predefined Macro	.c files	.h files	Predefined Macro	.c files	.h files	Predefined Macro	.c files	.h files
__LINE__	42.17	43.47	__TIME__	2.52	0.00	__STDC_IEC_559__	0.00	0.0
__FILE__	36.31	37.77	__STDC_VERSION__	0.00	0.00	__STDC_HOSTED__	0.00	0.0
__STDC__	15.77	18.11	__STDC_ISO_10646__	0.00	0.00			
__DATE__	3.23	0.65	__STDC_IEC_559_COMPLEX__	0.00	0.00			

__DATE__ macro

__DATE__ The date of translation of the preprocessing translation unit: a character string literal of the form **"Mmm dd yyyy"** , where the names of the months are the same as those generated by the **asctime** function, and the first character of **dd** is a space character if the value is less than 10. 2005

Commentary

translation phase 4

This string literal specifies the date when translation phase 4 executed, not when later phases of translation were executed (e.g., the output of this phase may be written to a file for subsequent processing on another day). A change of date during the translation process does not affect the character sequence contained in the string literal. The standard does not specify any accuracy requirements on the value returned by the __DATE__ macro.

predefined macros 2025 constant values during translation

Specifying that a space character is to be used if the day value is less than 10 the standard guarantees that the string literal always has the same width.

The discussion given for the __TIME__ macro is applicable here.

Common Implementations

Many translators unconditionally create the value to be used as the result of the __DATE__ macro once, when the translator is first started. This value is then used for all occurrences of __DATE__ during that translation. Translators generally rely on the accuracy of the date available in the environment in which they execute. Microsoft C supports the __TIMESTAMP__ macro as an extension. It expands to a string literal containing the date and time of the last modification of the current source file.

A single call to the `time` library function returns all of the information needed to create the values of the `__TIME__` and `__DATE__` macros.

2006 If the date of translation is not available, an implementation-defined valid date shall be supplied. __DATE__ date not available

Commentary

The C Standard cannot require that the environment in which a translator executes be capable of knowing the current date. However, it can require that a value always be supplied by the translator.

Common Implementations

The `time` library function, assuming this is how the translator obtains the current date, returns the value `(time_t)-1` if the current calendar time is not available.

2007 **`__FILE__`** The presumed name of the current source file (a character string literal).[152)] __FILE__ macro

Commentary

It is only the presumed name because the value may have previously been changed, earlier in the source file being translated, using a **`#line`** directive, or may have to be guessed (e.g., if the source was read from standard input). #line

Other Languages

Perl supports both the special variable `$PROGRAM_NAME` and the global special constant `__FILE__`.

2008 **`__LINE__`** The presumed line number (within the current source file) of the current source line (an integer constant).[152)] __LINE__ macro

Commentary

It is only the presumed name because the value may have previously been changed, earlier in the source file being translated, using a **`#line`** directive. #line

Other Languages

Perl supports the global special constant `__LINE__`.

2009 **`__STDC__`** The integer constant **`1`**, intended to indicate a conforming implementation. __STDC__ macro

Commentary

The macro `__STDC__` allows for conditional translation on whether the translator claims to be standard-conforming. It is defined as having the value 1. Future versions of the Standard could define it as 2, 3, etc., to allow for conditional compilation on which version of the Standard a translator conforms to. The C89 Committee felt that this macro would be of use in moving to a conforming implementation. Rationale

C++

Whether `__STDC__` is predefined and if so, what its value is, are implementation-defined. 16.8p1

It is to be expected that a C++ translator will not define the **`__STDC__`**, the two languages are different, although a conforming C++ translator may often behave in a fashion expected of a conforming C translator. Some C++ translators have a switch that causes them to operate in a C compatibility mode (in this case it is to be expected that this macro will be defined as per the requirements of the C Standard).

Common Implementations

Some implementations define this macro to have the value 0 when operating in non-standards conforming mode and sometimes the value 2 when extensions to the standard are enabled.

Coding Guidelines

The definition of this macro represents two pieces of information— (1) is definition as a macro, and (2) the tokens in its replacement list. Given that many vendors have chosen to always define the `__STDC__` macro and use its replacement list to specify the details of the conformance status, it is the value of the replacement list rather than the status of being defined that provides reliable information.

Cg 2009.1

Tests of the conformance status of a translator, in source code, shall use the value of the replacement list of the `__STDC__` macro, not its status as a defined macro.

__STDC_HOSTED__ macro

`__STDC_HOSTED__` The integer constant **`1`** if the implementation is a hosted implementation or the integer constant **`0`** if it is not. 2010

Commentary

The ability of an implementation to specify an accurate definition of the `__STDC_HOSTED__` macro depends on tight integration between various phases of translation.

C90

Support for the `__STDC_HOSTED__` macro is new in C99.

C++

Support for the `__STDC_HOSTED__` macro is new in C99 and it is not available in C++.

Common Implementations

While some vendors only sell implementations targeted to purely a hosted or freestanding environment, a few vendors sell into both markets.

Coding Guidelines

Support for the `__STDC_HOSTED__` macro is new in C99 and it is too early to tell whether there are any common translation phase dependency mistakes made by developers in its usage.

__STDC_VERSION__ macro

`__STDC_VERSION__` The integer constant **`199901L`**.[153)] 2011

Commentary

Because of existing, implementation, practice of giving values other than 1 to the `__STDC__` macro, existing code (and implementations) would have been broken had this macro been used to denote the version of the standard supported. The simplest solution was to define a new macro that explicitly indicated the version of the standard supported by an implementation.

The integer constant 199901 has type **`int`** in some implementations. Specifying a suffix ensures the type of the integer constant denoted by this macro is always the same.

C90

Support for the `__STDC_VERSION__` macro was first introduced in Amendment 1 to C90, where it was specified to have the value `199409L`. In a C90 implementation (with no support for Amendment 1) occurrences of this macro are likely to be replaced by 0 (because it will not be defined as a macro).

#if identifier replaced by 0

C++

Support for the `__STDC_VERSION__` macro is not available in C++.

Other Languages

Perl supports the special variable `$PERL_VERSION`.

2012 **__TIME__** The time of translation of the preprocessing translation unit: a character string literal of the form **"hh:mm:ss"** as in the time generated by the **asctime** function.

__TIME__ macro

Commentary

There is no guarantee that the value for the __TIME__ macro will be obtained on the same day as the value for the __DATE__ macro. It is possible, for instance, for the value __TIME__ macro to be obtained first (returning "23:59:59") followed by the value of the __DATE__ macro (returning a date on the following day).

The discussion given for the __DATE__ macro is applicable here.

2005 __DATE__ macro

Common Implementations

The extent to which host platforms on which the translators execute maintains an accurately time of day can vary significantly. The realtime clock in many PCs often drifts by several seconds in a day. The internet time protocol, nntp, provides one method of ensuring that the error in the estimated current time does not become too large over long periods.

Coding Guidelines

A common use of the __TIME__ macro is to provide program build configuration management information (e.g., the time at which a source file was translated). As such great accuracy is rarely required. During development there may be many builds and build times becomes important if more than one is performed in a day. Issues such as the values of the macros __TIME__ and __DATE__ being synchronized to refer to the same day is a configuration management issue and is outside the scope of these coding guidelines.

2013 If the time of translation is not available, an implementation-defined valid time shall be supplied.

Commentary

The discussion given for the __DATE__ macro also applies here.

2006 __DATE__ date not available

2014 The following macro names are conditionally defined by the implementation:

Commentary

Specifying that these macro names are conditionally defined simplifies the writing of source that may need to be processed by C90 implementations (which are unlikely to have defined these macro names). The affect on program conformance status of using conditional features is discussed elsewhere.

footnote 2

C90

Support for conditionally defined macros is new in C99.

C++

Support for conditionally defined macros is new in C99 and none are defined in the C++ Standard.

Coding Guidelines

Like the __STDC__ macro these conditionally defined macros represent two pieces of information. However, these macros are not required to be defined by an implementation. The extent to which implementations will define these macros to have values other than those specified in the standard are not known. The very infrequent use of these macros (which may be because they are new, or because source that needs to make use of the information they provide are not yet common) a guideline recommendation similar to that given for the __STDC__ macro is not considered cost effective.

2009 __STDC__ macro

2009.1 __STDC__ check replacement list

2015 **__STDC_IEC_559__** The integer constant **1**, intended to indicate conformance to the specifications in annex F (IEC 60559 floating-point arithmetic).

__STDC_IEC_559__ macro

Commentary

The C Standard does not specify how conformance to the specification in annex F is to be measured.

C90

Support for the __STDC_IEC_559__ macro is new in C99.

C++

Support for the __STDC_IEC_559__ macro is new in C99 and it is not available in C++.

The C++ Standard defines, in the std namespace:

18.2.1.1
```
static const bool is_iec559 = false;
```

false is the default value. In the case where the value is **true** the requirements stated in C99 also occur in the C++ Standard. The member is_iec559 is part of the numerics template and applies on a per type basis. However, the requirement for the same value representation, of floating types, implies that all floating types are likely to have the same value for this member.

---

footnote 151

151) See "future language directions" (6.11.9). 2016

---

footnote 152

152) The presumed source file name and line number can be changed by the **#line** directive. 2017

Commentary

This footnote clarifies the intent that implementations behave as if there is an association between evaluating a **#line** directive and the expanded form of subsequent occurrences of the __FILE__ __LINE__ macros.

C90

This observation is new in the C99 Standard.

C++

Like C90, the C++ Standard does not make this observation.

---

footnote 153

153) This macro was not specified in ISO/IEC 9899:1990 and was specified as **199409L** in ISO/IEC 9899/AMD1:1995. 2018

Commentary

__STDC_VERSION__ macro 2011

This is a brief history of the __STDC_VERSION__ macro.

The intention is that this will remain an integer constant of type **long int** that is increased with each revision of this International Standard. 2019

Commentary

standard integer types

The type **long int** (or its unsigned version) is the only standard integer type supported by the C90 Standard and required to be capable of supporting the necessary range of values.

Coding Guidelines

operator () Future language directions

In C99 the C committee made changes to the behavior, required by the C90 Standard, of strictly conforming programs. What behavior future revisions of the C Standard will specify is unknown, although the subsection on future language directions gives some warnings.

This macro is only really of practical use in checking that the translator being used supports a version of the C Standard that is greater than, or equal to, some known version.

__STDC__IEC_559_COMPLEX__ macro

__STDC_IEC_559_COMPLEX__ The integer constant **1**, intended to indicate adherence to the specifications in informative annex G (IEC 60559 compatible complex arithmetic). 2020

Commentary

__STDC_IEC_559__ macro 2015

The C Standard does not specify how adherence to the specification in annex G is to be measured (or how adherence differs from conformance).

C90

Support for the `__STDC_IEC_559_COMPLEX__` macro is new in C99.

C++

Support for the `__STDC_IEC_559_COMPLEX__` macro is new in C99 and is not available in C++.

2021 **`__STDC_ISO_10646__`** An integer constant of the form **`yyyymmL`** (for example, **`199712L`**) ~~,~~.

__STDC_ISO_10646__ macro

Commentary

The wording was changed by the response to DR #273. ISO 10646

C90

Support for the `__STDC_ISO_10646__` macro is new in C99.

C++

Support for the `__STDC_ISO_10646__` macro is new in C99 and is not available in C++.

Other Languages

The few languages (e.g., Java) that currently support ISO 10646 do not provide access to equivalent version information. ISO 10646

Common Implementations

The implementation shipped with RedHat Linux version 9 has the value 200009L.

2022 If this symbol is defined, then every character in the Unicode required set, when stored in an object of type **`wchar_t`**, has the same value as the short identifier of that character.

Commentary

This is a requirement on an implementation that defines the `__STDC_ISO_10646__` macro. A short identifier is representable in eight hexadecimal digits (i.e., values in the range `0..4294967295`). For instance: short identifier

```
1  L'\u00A3' == L'\x00A3'            if __STDC_ISO_10646__ defined
2  L'\u00A3' == L'£'                 always true
3  L'\x00A3' == (wchar_t) 0x00A3     always true
4  L'\x00A3' == L'£'                 if __STDC_ISO_10646__ defined
5  L'£' == (wchar_t) 0x00A3          if __STDC_ISO_10646__ defined
```

The term *Unicode required set* is defined in the following C sentence.

This wording was changed by the response to DR #273.

C90

This form of encoding was not mentioned in the C90 Standard.

C++

This form of encoding is not mentioned in the C++ Standard.

Unicode required set

2023 The *Unicoderequiredset* consists of all the characters that are ~~intended to indicate that values of type **wchar_t** are the coded representations of the characters~~ defined by ISO/IEC 10646, along with all amendments and technical corrigenda, as of the specified year and month.

Commentary

This gives permission for implementations to support amendments and technical corrigenda to ISO/IEC 10646 that are published after the C99 Standard. However, all documents published on or before the specified month must be supported. ISO 10646

The wording was changed by the response to DR #273.

Coding Guidelines

Program dependencies on the version of ISO/IEC 10646 is a configuration management issue that is outside the scope of these guidelines.

__STDC_MB_MIGHT_NEQ_WC__ The integer constant **1**, intended to indicate that, in the encoding for **wchar_t**, a member of the basic character set need not have a code value equal to its value when used as the lone character in an integer character constant. 2024

Commentary

L'x' == 'x'
basic character set
wide character

Until the publication of TC2, implementations were required to support the equality `'x' == L'x'`, where x is any member of the basic character set. This requirement restricted an implementation's choice of encoding for the type `wchar_t`. The response to DR #279 removed this restriction, but did not provide a mechanism for developers to write code that checked the behavior of their implementation (a large body of existing code relies on pre-TC2 guaranteed behavior).

The expression `L'\xhh' == '\xhh'` is always true (as long as `0xhh` is less than `UCHAR_MAX`). Developers would be very confused if this equality was true for escape sequences, but was not true when the escape sequences were replaced by members of the basic execution character set having the same numeric value (in a particular character set, for instance Ascii).

This sentence was added by the response to DR #333 and provides a specification for the pre-defined macro name introduced by the response to DR #321 (and supported by the Austin Group).

---

predefined macros
constant values during translation

The values of the predefined macros (except for **__FILE__** and **__LINE__**) remain constant throughout the translation unit. 2025

Commentary

This is a requirement on the implementation. The standard places no requirements on exactly when the implementation assigns a value to the `__DATE__` and `__TIME__` macros.

The predefined macros `__DATE__` and `__TIME__` are intended to provide configuration management information about when the source was translated. Having to deal with time differences caused by the finite time needed to translate a source file would be an unnecessary complication for developers.

Given a sufficiently high-performance processor and fast translation, or a translation environment where no date and time information is available, it is possible that the values of the `__DATE__` and `__TIME__` macros will be the same during translations of different source files.

Common Implementations

Many implementations allow more than one source file to be specified, for translation, at the same time. The standard is silent about the values of these predefined macros, across multiple sources files, during a single invocation of a translator.

program transformation mechanism
footnote 5

Many implementations use several programs to translate a source file, preprocessing being handled by one of these programs. The details of invoking these separate programs is handled by a controlling program that provides the user interface, including breaking the translation of multiple source files into a sequence of translations of individual source files. Given this implementation strategy it is very likely that the preprocessor will be invoked separately for every source file being processed (with the value of the predefined macros being set during the startup of this preprocessor program) and can be different between different invocations.

---

predefined macros
not #defined

None of these macro names, nor the identifier **defined**, shall be the subject of a **#define** or a **#undef** preprocessing directive. 2026

Commentary

This wording covers some of the possible source code constructs where **defined** can occur, but not all. For instance:

```
#define f(defined) defined

x = f(1);

#if f(1)
/* ... */
#endif

#if defined(defined)
/* ... */
#endif
```

If the identifier **defined** were to be defined as a macro, `defined(X)` would mean the macro expansion in C text proper and the operator expression in a preprocessing directive (or else that the operator would no longer be available). To avoid this problem, such a definition is not permitted (§6.10.8). Rationale

C++

The C++ Standard uses different wording that has the same meaning.

If any of the pre-defined macro names in this subclause, or the identifier ***defined****, is the subject of a* ***#define*** *or a* ***#undef*** *preprocessing directive, the behavior is undefined.* 16.8p3

Common Implementations

Some, but not all, implementations allow predefined macros to be **#undef**ed and for the identifier **defined** to be defined as a macro name.

Coding Guidelines

The usage described is very rare and for this reason a guideline recommendation is not considered cost effective.

2027 Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore. macro name predefined reserved

Commentary

This is a requirement on the implementation. It also acts as a warning to developers that macro names beginning with these sequences of characters may be reserved by an implementation. A predefined macro is generally considered to be one that is defined by a translator prior to the start of translation, i.e., it is not necessary for the source file to **#include** a particular header for it to be defined.

C++

The C++ Standard does not reserve names for any other predefined macros.

Common Implementations

The definition of some predefined macros may be affected by command line options passed to the translator. For instance, specifying the version of a particular processor architecture to generate machine code for. Such version specific architecture macro definitions might then affect which conditional inclusion directives are processed, within implementation supplied headers.

Although the standard reserves identifiers having these spellings for use by implementations, most implementations also use identifiers having spellings outside of this set, for their own internal use.

2028 The implementation shall not predefine the macro **__cplusplus**, nor shall it define it in any standard header. __cplusplus

Commentary

This is a requirement on the implementation. It has become established practice to use this macro name to distinguish those parts of a header that are specific to C++ only and should be ignored by a C translator.

C90

This requirement was not specified in the C90 Standard. Given the prevalence of C++ translators, vendors were aware of the issues involved in predefining such a macro name (i.e., they did not do it).

C++

16.8p1 *The name __cplusplus is defined to the value 199711L when compiling a C++ translation unit.*[143)]

Other Languages

This macro name is unique to C in the sense that this language is sufficiently similar to the C++ language for implementors to be able to share headers in implementations of both languages.

Forward references: the `asctime` function (7.23.3.1), standard headers (7.1.2). 2029

References