ISO/IEC JTC 1/SC 22/WG14 **N 2175**

September 30, 2017 **v 1**

Syntax proposal for attributes in C2x
accomodate C and C++?

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

We propose to find a middle ground between C and C++ to resolve the syntax compatibility problem for attributes. Our approach uses Unicode characters to resolve the issue: special parenthesis are introduced as tokens and a new separator character is used to separate the prefix (**std** or implementation defined) from the attribute name. We use code points with glyphs that resemble the C++ usage of `[[...]]` and `::` such that visual habits are respected.

1. INTRODUCTION

The integration of attributes into the C language has been discussed since over 10 years, now. A quick scan on WG14's document site turns up at least the following documents:

N2165	2017/09/20	Ballman	Attributes in C (revision to N2137)
N2137	2017/03/13	Ballman	Attributes in C (revision to N2049)
N2053	2016/07/28	Ballman	The maybe_unused attribute
N2052	2016/07/28	Ballman	The fallthrough attribute
N2051	2016/07/28	Ballman	The nodiscard attribute
N2050	2016/07/28	Ballman	The deprecated attribute
N2049	2016/07/28	Ballman	Attributes in C
N1403	2009/09/28	Svoboda	Towards support for attributes in C
N1297	2008/03/14	Stoughton	Attribute Syntax
N1280	2008/02/11	Walls	Attribute Names, Use Existing Practice
N1279	2008/01/23	Walls	Attributes Commonly Found in Open Source Applications
N1273	2007/10/11	Stoughton	Attributes
N1262	2007/09/10	Kwok	Towards support for attributes in C++
N1259	2007/09/09	Myers	C1x: Attribute syntax
N1233	2007/04/26	Wong	Towards support for attribute-like syntax

Table I. WG14 documents concerning the introduction of attributes.

Unfortunately, the documents in that list don't agree upon the syntax that should be applied for attributes, would they be introduced to the C language.
On the other hand, integrating the functionality into C seems to be largely consensual within WG14. This proposal tries to break the blockage by proposing a new way to integrate attributes smoothly, without needing to fallback to "uglification" of identifiers (such as in **_Bool** or **_Thread_local**).
This proposal concerns a feature (attributes) that are widely present in existing implementations, but with varying syntax. The syntax proposed here is an *invention by necessity* that tries to consolidate these different existing C implementations, portability requirements that are specific to the C language, and the corresponding feature of C++.

2. PROBLEM DESCRIPTION

The principal idea of attributes is to introduce a syntactic tool to the C language that

— allows to add specifications to existing language constructs,
— has clear semantics to which construct it applies,
— is backwards compatible, that is for which the addition to headers will not invalidate existing code, nor change its semantics.

Additionally, it would be highly desirable if the new syntax would also seamlessly integrate into existing compilers and toolkits.
C++ already has adapted a syntax that (in the context of C++) fits well with these goals, namely double brackets `[[ ]]` to introduce attributes and `::` to separate name components

of the attribute names. Aaron Ballman's proposal in document N2165 shows that its model of application to language constructs is sound and fits well for C, too.
Unfortunately though, the syntax does not integrate well with C. There are two issues:

— The token sequence `]]` appears in valid code, so parsers must be adapted to treat that special case.
— Attribute specifications of the form `some::thing` where `some` and `thing` are non-reserved identifiers can interfere badly with existing code.

Whereas the first point is relatively minor and poses the burden "only" on compiler implementors, the second is much more serious and concerns the whole C code base.
The reason for this is that there is no good solution for C how the character sequence `::` could be integrated. It could

— remain a two token sequence as it is now, or
— become a token of its own right, and thus be inseparable.

Regardless, a construct such as `some::thing` exposes two non-reserved identifier tokens (`some` and `thing`) to the preprocessor. Thus such attributes would collide with existing identifiers (user space and C library).
Standard C itself would certainly not rush to add many **`std`** attributes, so the amount of additional identifiers that had to be reserved only for that would be small, namely **`std`** and the list of names of such attributes.
The real problem arises for those attributes that are the main purpose of the attribute extension: implementation specific attributes. As an example the GNU compiler framework documents the following attributes that are common to all supported architectures:

```
 alias aligned alloc_align alloc_size always_inline artificial assume_aligned
 bnd_instrument bnd_legacy cleanup cold common constructor const deprecated
 destructor error externally_visible flatten format_arg format gnu_inline hot
 ifunc interrupt_handler interrupt leaf malloc mode no_address_safety_analysis
no_icf no_instrument_function no_reorder no_sanitize_address no_sanitize_thread
 no_sanitize_undefined no_split_stack no_stack_limit noclone nocommon noinline
nonnull noplt noreturn nothrow optimize packed pure returns_nonnull returns_twice
 section sentinel simd stack_protect target_clones target tls_model unused used
vector_size visibility warn_unused_result warning weakref weak
```

(And *a lot* more that are architecture specific.)

For their own attribute syntax, GNU uses *uglification_ by_ underscores*™ to protect against naming conflicts. E.g a valid GNU function attribute would be `__attribute__((__malloc__))`. Translated into C++ attributes this gives `[[gnu::malloc]]`.

— The particle `gnu` may conflict with a user space identifier, in particular a macro.
— The particle **`malloc`** may conflict with a macro in the C library.

> *The number of interactions with existing code that implementation defined attribute names may have is inpredictable, and could turn out to be a real burden for porting software forward to new compiler versions or architectures.*

3. POSSIBLE SOLUTIONS

3.1. Uglification

Traditionally, if C introduces new keywords to the language it choses from its reserved identifier space, that is identifiers that

— start with an underscore followed by another underscore or a captial Latin letter.

This lead e.g to the introduction of **_Bool** and **_Static_assert**.
Additions to the library are made by reserving a name prefix, *e.g* **atomic_**, E, **SIG** or **memory_order_**.

3.1.1. Reserved identifiers. Reserved identifiers with leading underscores and capital letter or even all-caps could be used for attribute name particles, e.g [[_Std::_Maybe_unused]], [[_Gnu::_Malloc]], [[_STD::_MAYBE_UNUSED]], [[_GNU::_MALLOC]].
This would already be far from the existing C++ usage and would not much ease the design of common C and C++ interfaces.

3.1.2. Leading double underscores. Leading underscores could be used for attribute name particles, e.g [[__std::__maybe_unused]] or [[__gnu::__malloc]].
This is a bit closer to the existing C++ habits, but constructs such as [[__gnu::__format(__printf, 2, 3)]] quickly become lengthy and illegible.

3.1.3. Prefixes. Prefixes could be a solution for standard attributes. We could e.g use std_ and have [[std_maybe_unused]] etc.
This would not be an easy solution for implementation specific attributes, though, because no programmer could reasonably foresee which compilers could be relevant for their code in the future, nor would there be a central register of compiler prefixes that they could easily consult.

3.2. Introduce poor-man's namespaces to C

The :: construct works well for C++ for two reasons. First, because there it refers to a well established feature, namely identifier separation by class names or namespaces. Second, the preprocessor is used much less than in C and so conflicts with user defined macros will be much less frequent.
Introducing such a feature to simple C compilers would be a hurdle, because the grammar would not only change for the newly introduced tokens, but also for the introduction of names that are structured hierarchically.
In particular, to avoid the name clash with macros it would require to change the lexing phase (phase 3) that would take care of the composed identifiers. Still, internally it would have to mark the position of the :: inside the identifier with some special character.

4. OUR PROPOSAL: USE NEW CHARACTERS

We propose to circumvent all these problems by using Unicode code points to describe the features that we need to implement this syntactically:

— Use double bracket characters as in ⟦ ⟧ to introduce and terminate attribute specifcations.
— Use ∷ as a special separator character between the prefix and the name of an attribute.

This leads to things such as ⟦**std**∷maybe_unused⟧ or ⟦gnu∷**malloc**⟧, that are visually equivalent to the corresponding C++ syntax.
The most important features of this proposal to avoid naming conflicts are:

(1) *Introduce a new separator character* ∷ *that may not be part of any other identifier.*
(2) *Define attributes as composed identifiers consisting of a prefix and an ordinary identifier.*
(3) *Join these two parts with* ∷ *to form a valid attribute name.*

4.1. Most mathematical symbol code points are free to use

Almost unnoticed, Unicode support as of \u and \U had been added to C99. They allow to include any Unicode point in wide character constants and strings and to extend the usable character set for identifiers in a portable way. Thereby

```
1  #define \u03c0 3.14159265358979323846
```

is a valid construct, even if on some specific platform π is not present in the source character set.
C allows a wide range of code points to appear in identifiers, but there are some restrictions, namely for code points that Unicode categorizes as punctuation or mathematical symbols. Valid C code may only contain such characters inside character constants or strings:

> *No C program may contain code point \u2237 in identifiers.*

> *The behavior of a C program containing code points \u27E6 or \u27E7 outside character constants, strings or comments is undefined.*

Therefore we can propose the use of characters ⟦ ⟧ ∷ for attributes, see Table II. Since the characters are previously unused in C, syntax can easily be adapted to integrate the new feature.

glyph	code point	category	grammar	optional	digraph	trigraph
∷	\u2237	math symbol	identifier			??::
⟦	\u27E6	math, open parenthesis	token	[[	<::	
⟧	\u27E7	math, closing parenthesis	token	]]	::>	

Table II. Code points used for attributes

4.2. Provisions for a seamless transition

The integration of this proposal into most implementations should be easy. Modern platforms are much better in integrating different scripts to display Unicode code points than this had been the case in the beginning of C. Many, if not most, are able to use UTF-8 encoded source files, and easily present all characters of Unicode's *Basic Multilingual Plane*, which includes the characters proposed here.
Also, nowadays text editors and IDEs are easily extensible such that they permit insertion of arbitrary characters, be it from a drop down menu, by typing the hex sequence or the name of the character or by defining an editor macro.
So the integration of attributes that are encoded with these characters into modern environments is easy most of the time. Also, at the beginning attributes will principally occur in header files, so the problem will be mostly for implementers and how to provide headers with patched up specifications.
If nevertheless, implementations have difficulties because one of the three characters are not present in the source character encoding, they can fallback to different presentations:

— All encodings that provide a \ character can use the universal characters encoding, \u2237, \u27E6 or \u27E7.
— Otherwise, a ∷ replacement must be performed early in translation phase 1, that is when trigraphs are replaced. One possibility would be to use the unused trigraph sequence ??:. A variant that would be be closer to the C++ visual would be to replace ??::.
— Replacements for ⟦ and ⟧ only need to be introduced for translation phase 3, tokenization.

— The ⟧ character cannot be encoded by a]] digraph without invalidating currently valid C code. We propose to allow this as an implementation defined property. As a general replacement, We introduce "digraph" ::>.
— The ⟦ character may be encoded by a [[digraph, because this character sequence cannot appear in valid code other than strings etc. Nevertheless for the symmetry with]] we prefer to introduce "digraph" <::.

As an example, the following two declarations

```
1  ⟦some::thing⟧ int A[];
2  ⟦gnu::malloc⟧ void* realloc(void*, size_t n);
```

can be written as follows:

```
1  \\u27E6some\\u2237thing\\u27E7 int A[];
2  <::gnu??::malloc::> void* realloc(void*, size_t n);
```

Whereas the first line is barely readable, the second could be a realistic alternative during a transition to C2x.

5. SUGGESTED ADDITIONS TO THE C STANDARD

This only handles modifications that will be necessary for the naming aspect of this proposal. For the modifications needed to introduce attributes *per se*, see N2165.

(1) Introduce the new characters to 5.2.1 p3. Introducing new characters to the base character set it not possible since this would imply one-byte encodings for the new characters, which we can't assume.

> *In the extended source and execution character sets, there shall be the* additional graphic characters *corresponding to the universal characters with the following short identifiers and glyphs:*
>
> ```
> 2237 ∷        27e6 ⟦          27e7 ⟧
> ```
>
> *Whether these characters have one byte or multibyte encodings is implementation defined.*

(2) Make sure that the new characters are treated the same, regardless how they are encoded in the source.

> *The additional graphic characters have specific roles in the grammar, notwithstanding if they are entered verbally or through the universal character name construct.*

(3) Add the optional "trigraph" ??:: to 5.2.1.1.

> *In addition, it is implementation defined whether or not the character sequence* ??:: *is replaced by the character* ∷.

(4) Add ∷ to "identifier-nondigit" in 6.4.2.1 p1, Syntax.
(5) Add a new subclause 6.4.2.3 that describes reserved identifiers:

> ***6.4.2.3 Reserved identifiers***
> ***Constraints***
> *Identifiers starting with an underscore followed by another underscore or a capital letter and identifiers that contain the character* ∷ *are reserved for future extensions of the language and the library, and for implementation specific*

> *purpose. Implementations shall not use identifiers starting with* **std**::, **stdc**:: *or* **__STDC_** *other than those that are defined by this international standard.*

(6) Add ⟦, ⟧, <:: and ::> as punctuation to 6.4.6 p1.
(7) Add <:: and ::> as digraphs to 6.4.6 p3.
(8) Add 2237 to Annex D1.