Proposal for C2x
WG14 N2493

Title:	What we think we reserve
Author, affiliation:	Aaron Ballman, GrammaTech
Date:	2020-02-21
Proposal category:	Modifying existing normative requirements
Target audience:	Users

Abstract: The C standard normatively reserves identifiers and some of the reservations impose onerous requirements on programmers. The most severe requirements are generally unknown to programmers, not checked by tools, and demonstrate a disconnect between the C standards committee and the language as it is used by programmers.

Reply-to: Aaron Ballman (aaron@aaronballman.com)
Document No: N2493
Revises Document No: N2409
Date: 2020-02-21

Summary of Changes

N2493

- Switched to the idea of a potentially reserved identifier

N2409

- Original proposal

Introduction and Rationale

C does not have a syntactic feature for reserving identifiers. Instead, the standard makes sweeping identifier reservations using lexical patterns, such as identifiers starting with an underscore followed by an uppercase letter, and the C committee expects the entire C community to know and adhere to the reservations to avoid breaking code when adding new language or library features. However, some of the current reservations result in onerous requirements on the programming community that are often not reliably checked by implementations and tools, or honored by users.

What makes a reserved identifier?

Identifier reservations are unfortunately split into two different places within the standard. 7.1.3p1 gives what looks to be an exhaustive list of reserved identifiers, and 7.1.3p2 goes on to state: No other identifiers are reserved. However, you need to read p1 carefully to note that 7.31 Future Library Directions also includes a list of reserved identifiers that are reserved under entirely different circumstances. For instance, 7.1.3 talks about reserving identifiers only if their associated header is included, while 7.31p1 reserves identifiers regardless of what headers are included (if any).

7.1.3 Reserved Identifiers

Identifiers with two leading underscores or a leading underscore followed by a capital letter. However, this only applied in cases where the identifier is not lexically identical to a keyword.

Reserved	Unreserved
`int __foobar, _Foobar`	`#define _Generic(x)`

Identifiers that begin with an underscore at file scope.

Reserved	Unreserved
`int _foobar;`	`int func(void) { int _foobar; }`

Macro names and identifiers with external linkage that are specified in the C standard library clauses.

Reserved	Unreserved

```
#include <locale.h>
int func(void) {
  const char *localeconv;
}
```

```
int func(void) {
  const char *localeconv;
}
```

This proposal does not propose any changes to these reserved identifiers.

7.31 Future Library Directions

The individual reservations make claims as to what kinds of identifiers are reserved (macro names, function names, etc.) and what header file is impacted. However, p1 makes it clear that all identifiers reserved from this subclause are reserved identifiers regardless of what header files are included, meaning that these rules apply to all C code. Further, reserving an identifier pattern for a given use has limited practical effect on the context under which the identifier is reserved. Reserving an identifier for any use effectively reserves it for all uses in a practical sense. For instance, reserving something for use as a macro name or enumeration constant practically ensures that the name cannot also be used as the identifier in a function declaration, and vice versa. In effect, these identifiers are reserved for all uses in C regardless of what header files (if any) are included, and so the identifier reservations are being listed below by pattern rather than by header or entity.

- is, to, str, mem, wcs, atomic_, memory_, memory_order_, cnd_, mtx_, thrd_, or tss_ followed by a lowercase letter
- E, FE_, LC_, SIG, SIG_, ATOMIC_, or TIME_ followed by an uppercase letter
- E followed by a number
- PRN or SCN followed by a lowercase letter or the uppercase letter X
- Identifiers starting with uint or int and ending with _t, or UINT or INT and ending with _MAX, _MIN, or _C
- cerf, cerfc, cexp2, cexpm1, clog10, clog1p, clog2, clgamma, ctgamma, optionally followed by f or l

While many of these reservations seem reasonable or even necessary, they have some far-reaching consequences for introducing undefined behavior in user programs. Consider the following examples:

```
enum structure { // reserved
  isomorphic, // reserved
  nonisomorphic
};
void memorize_secret( // reserved
  const char *string // reserved
);
struct toxicology { // reserved
  enum condition {
    cnd_clean, // reserved
    cnd_dirty  // reserved
  } cnd;
};
#define ENTOMOLOGY 1 // reserved
#define SIGNIFICANT_RESULTS 1 // reserved
#define TIME_TO_EAT 1 // reserved
#define ATOMIC_WEIGHT .000001f // reserved
#define INTERESTING_VALUE_MIN 0 // reserved
```

While these identifiers may seem contrived, it does not stretch the imagination to believe that programmers will accidentally use reserved identifiers with relative frequency without realizing it. A survey of the most egregious prefix patterns demonstrates that there are a considerable number of English words prohibited from use in C currently: e (33,921 words), to (3,810 words), is (3,267 words), str (1,643 words), sig (470 words), and mem (231 words). A survey of compilers and static analyzers were unable to identify a single tool warning users about all forms of reserved identifiers, including ones from the Future Library Directions, though all of the tools surveyed were able to warn about varying subsets of the reserved identifiers. The tools surveyed were: Clang, Microsoft Visual Studio, GCC, ICC, CodeSonar, CppCheck, Coverity, QAC, and two unnamed static analysis tools (not all tools can be listed by name due to Terms of Service requirements).

Code in the Wild

Code in the wild seems to ignore the reservations from 7.31. It is trivial to find examples of identifiers in popular C projects that violate the reserved identifiers restrictions from 7.31. A brief survey of a few popular C projects doing a simple regular expression search over header files finds the following examples, with the reserved identifiers highlighted in red for clarity:

sqlite (https://sqlite.org/index.html)

```
int (*strlike)(const char*,const char*,unsigned int);
#define EP_Reduced    0x002000 /* Expr struct EXPR_REDUCEDSIZE bytes only */
void *token;                   /* id that may be used to recursive triggers */
```

Windows 10 SDK (https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk)

```
#define ERROR_SUCCESS                    0L
typedef struct tagRECT
    {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
    }       RECT;
```

ReactOS (https://github.com/reactos/reactos)

```
extern int  iso9660_level;
extern int  iso9660_namelen;
struct directory_entry {
      …
      unsigned int      total_rr_attr_size;
      …
};
struct chmcTopicEntry {
      UInt32 tocidx_offset;
      …
};
#define POW2(stride) (!((stride) & ((stride)-1)))
```

libuv (https://github.com/libuv/libuv)

```
#define container_of(ptr, type, member) \
  ((type *) ((char *) (ptr) - offsetof(type, member)))
typedef enum {
  TCP = 0,
  UDP,
  PIPE
} stream_type;
# define ENABLE_EXTENDED_FLAGS 0x80
```

libiconv (https://www.gnu.org/software/libiconv/)

```
static inline int streq8 (const char *s1, const char *s2, char s28);
#define isxbase64(ch) ((ch) < 128 && ((xbase64_tab[(ch)>>3] >> (ch&7)) & 1))
#define EXPR_SIGNED(e) (_GL_INT_NEGATE_CONVERT (e, 1) < 0)
```

Proposal

The goal of the future language and library reservations is to alert C programmers of the potential for future standards to use a given identifier as a keyword, macro, or entity with external linkage so that WG14 can add features with less fear of conflict with identifiers in user's code. However, the mechanism by which this is accomplished is overly restrictive – it introduces unbounded runtime undefined behavior into programs using a future language/library reserved identifier despite there not being any actual conflict between the identifier chosen and the current release of the standard. While it may be appealing to ignore this as "harmless" undefined behavior because implementations would not change runtime behavior to benefit from this latitude, it does still pose a burden for users. For instance, coding standards will often have a blanket prohibition against instances of undefined behavior (such as Rule 1.3 in MISRA C:2012 [0] or MSC15-C in the CERT C Secure Coding Standard [1]).

Instead of making the future language/library identifiers be reserved identifiers, causing their use to be runtime unbounded undefined behavior per 7.1.3p1, we propose introducing the notion of a *potentially reserved identifier* to describe the future language and library identifiers (but not the other kind of reservations like `__name` or `_Name`). These potentially reserved identifiers would be an informative (rather than normative) mechanism for alerting users to the potential for the committee to use the identifiers in a future release of the standard. Once an identifier is standardized, the identifier stops being potentially reserved and becomes fully reserved (and its use would then be undefined behavior per the existing wording in C17 7.1.3p2). These potentially reserved identifiers could either be listed in Annex A/B (as appropriate), Annex J, or within a new informative annex. Additionally, it may be reasonable to add a recommended practice for implementations to provide a way for users to discover use of a potentially reserved identifier. By using an informative rather than normative restriction, the committee can continue to caution users as to future identifier usage by the standard without adding undue burden for developers targeting a specific version of the standard.

It is worth noting that some of the reserved identifiers in the Future Library Directions subclause are reserved for use by the implementation rather than solely for potential future standardization. Such an identifier would continue to be a reserved identifier rather than converted to a potentially reserved identifier.

Acknowledgements

I would like to recognize the following people for their help in this work: Alex Gilding, Tom Honermann, Christof Meerwald, Clive Pygott, Robert Seacord, and David Svoboda.

References

[0] MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems: March 2013. MIRA Limited, 2013.

[1] Seacord, Robert C. The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems. Addison-Wesley, 2014.