ISO/IEC JTC 1/SC 22/WG14 N 2129

March 8, 2017

v 1

Deprecate __LINE__

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

Because of the loophole for untyped function parameters, the usage of the `__LINE__` macro can lead to undefined behavior when used as integer literal. This papers reviews the possibilities to improve this situation in C2x and proposes to replace `__LINE__` by two new macros `__line__` (producing a string) and `__LINENO__` (producing a literal of type `uintmax_t`).

Introduction

The articulation between handling of numbers in the preprocessor and later compilation phases can lead to undefined behavior:

```
printf("visiting␣line␣%d\n", __LINE__);   // may have undefined behavior
```

This use of `__LINE__` is only well defined if the current line number is less or equal to `INT_MAX`. Otherwise `__LINE__` has type `long` or maybe even `long long`, see below. Passing a wider object to a `"%d"` format specifier has undefined behavior. In practice it can crash programs, open vulnerabilities or even provide opportunities to introduce backdoors.

Such an overflow may sound unlikely, but on restricted platforms `INT_MAX` may be as small as $2^{15} - 1 = 32767$. Automatically generated C code easily has more lines than this, or it may have `#line` directives with larger numbers.

Usages of `__LINE__` with values that exceed `INT_MAX` would be difficult to detect by code review.

1. PROBLEM DISCUSSION

Arithmetic in C's preprocessing and later compilation phases are only loosely coupled. The only constraint that C11 imposes on numbers when they are treated as such during preprocessing is that the arithmetic has to be performed as if these numbers where of type `[u]intmax_t`.

The problem here described is caused by the unfortunate combination of several loopholes in C's type system:

— un-prototyped function parameters, in particular for variadic functions,
— type adjustment of decimal literals to fit the first in `int`, `long` or `long long`,
— a discrepancy between the minimal requirement for `INT_MAX` ($2^{15}-1$), permitted numbers in `#line` directives (up to $2^{31} - 1$) and permitted numbers during preprocessing (up to $2^{63} - 1$).

The expansion of `__LINE__` may have type `long` or `long long`

The type of the expansion of `__LINE__` could be `int` or `long` for the reasons seen above, but it could even be forced to `long long` by malicious use of a `#line` directive. C11's section 6.10.4, p. 3, restricts line numbers to $2^{31} - 1 = 2147483647$ or less, but a line number then could grow beyond that value.

```
344  #define LARGE 2147483600   // not at LONG_MAX, innocent?

21476  /* ... much later in the source */
21477  #line LARGE "automatically-generated-please-ignore"

2147483646  /* ... even later inside a function ... */
```

```
2147483647    printf("%s␣%ld:␣a␣toto␣failure,␣please␣ignore\n",
-2147483648           __FILE__, __LINE__); // __LINE__ is 2147483648
```

If `long` has a width of 32 this will result in a number literal of type `long long`.[1]

Setting `#line` manually to an arbitrary large value would probably not pass thorough code review. But `#line` directives as demonstrated that are subject to macro expansion are easily overlooked.

The problem is difficult to diagnose

If the preprocessing phase is implemented as a separate program, `cpp`, say, there is no easy way to know from within that preprocessor program to which type a `__LINE__` macro expands in later compilation phases. So the preprocessing phase cannot easily be made aware of the discrepancy and we can't expect a diagnostic in that phase.

2. RECOMMENDABLE PRACTICE

Candidates for replacement of `__LINE__`

To avoid this problem, projects could prohibit the use of `__LINE__` other than in the following forms. All of them produce integer constant expressions (ICE) such that they should not incur runtime overhead. Two of them abort compilation if the value is too large.

```
// incompatible with existing usage of __LINE__
#define __LINE0__ (__LINE__+0LL)
// incompatible with existing usage of __LINE__
#define __LINE1__ INT32_C(__LINE__)
// incompatible with existing usage of __LINE__
#define __LINE2__ INTMAX_C(__LINE__)
// large number aborts compilation, not valid in #if, type?
#define __LINE3__ _Generic((char(*)[(__LINE__ <= INT32_MAX)+1])0, \
                           char(*)[2]: __LINE__)
// large number → print something ugly, needs prior knowledge
#define __LINE4__ ((int)(__LINE__ <= __MAGIC__ ? __LINE__ : -1))
// large number → print something ugly, implementation defined
#define __LINE5__ ((int)(__LINE__ <= INT32_MAX ? __LINE__ : -1))
// large number aborts compilation, not valid in #if
#define __LINE6__ _Generic(__LINE__, int: __LINE3__)
// large number → print something ugly, not valid in #if
#define __LINE7__ _Generic(__LINE__, int: __LINE5__, default: -1)
```

Advantages and disadvantages

Other than current usage might expect, `__LINE0__` to `__LINE3__` have the disadvantage that the resulting type is generally different from `int`. Therefore using one of these may require code changes in many places, namely in places where `__LINE__` is used as argument to `printf` or similar functions. Among these four, `__LINE1__` or `__LINE3__` are preferable, because they reflect the value restriction of the standard to $2^{31} - 1$. Other advantages are that `INT32_C(.)` is a no-op on many modern platforms (for `__LINE1__`) or that they abort compilation in case of larger value (for `__LINE3__`).

The remaining macros force their type to be `int`. Therefore they could be used in most places that previously used `__LINE__`, with the exception for `__LINE6__` and `__LINE7__` that are not compatible with preprocessor `#if`. `__LINE4__` and `__LINE5__` need prior knowledge about the platform: `__LINE3__` needs a magic number to take the right decision and `__LINE4__` leads to an implementation defined conversion from `long` to `int` if `int` has less than 32 bit.

[1] As we can see from the printout of that code, LaTeX' `listings` package is vulnerable to the same line number overflow.

In summary:

> *None of these is a full replacement for all practical uses of the __LINE__ macro.*

Personally, I have a slight preference for __LINE6__, because this aborts compilation when an unsuitable value is encountered.

3. FUTURE DIRECTIONS

This is not a defect report against C11. I was not able to come up with a consistent idea of how definitions of __LINE__ "could have been meant" that would avoid this problem. Also, I think this discussion should be included in the larger discussion about line numbers and the preprocessing phase, as they were started by some defect reports against C11.

Therefore I propose the following changes for C2x:

(1) Numbers of physical lines are of type uintmax_t, so they wrap around on overflow.
(2) Add a constant __LINENO_MAX__ of value at least $2^{31} - 1$ to <limits.h>.
(3) Add a specification that determines the line numbers of physical and logical source lines uniquely.
(4) Constrain any line number that is subject to a usage of __LINE__ (or equivalent) to be less than or equal to __LINENO_MAX__.
(5) Deprecate the use of __LINE__.
(6) Provide macros with similar functionality that avoid the vulnerability.

Replacements for __LINE__

Possible replacements could work conceptually as if they were defined as

```
// similar to __func__
#define STRINGIFI(X) #X
#define STRINGIFY(X) STRINGIFI(X)
#define __line__ STRINGIFY(__LINE__)
// incompatible with existing usage of __LINE__
#define __LINENO__ UINTMAX_C(__LINE__)
```

The most important would be __line__, the stringified expansion of __LINE__, because probably the most important usage of __LINE__ in applications is to output the line number for debugging purposes. The use of __line__ avoids any runtime overhead of number conversion. Since the result is a string literal, it can even become integral part of a printf format string, without passing through a "%s" conversion.

__LINENO__ could be used in cases where the number is needed for arithmetic or where it is to be concatenated with another token to compose a specialized identifier. The forced type of [u]intmax_t could help such that the use is consistent with in #if directives.

Uniqueness

If bounds checking for printing line numbers is made a constraint, we should guarantee that such a constraint is triggered portable between platforms that use the same __LINENO_MAX__. Therefore we should establish a rule that determines the line number that is returned by __LINE__ (or __line__ and __LINENO__) uniquely. Currently there are borderline cases for invocations that appear in the same logical source line.

A logical source line L can be the concatenation of a set of physical source lines $P_0, \ldots, P_r$. As soon as the preprocessing phase starts the processing of line L, and even before it knows the full extent of L, it knows the starting line of the sequence P_0. Thus, the simplest and less restrictive rule is to impose that the line number that is attributed to L is P_0.