# Issues Affecting a `long long` Data Type

WG14/N497 X3J11/95-098
1995-12-22

*David Keaton*
*dmk@dmk.com*

This document presents some issues to be considered before adding a `long long` data type to the C language. Adding this type affects the longest integral type available, the constant promotion rules, and portability.

## Longest Integral Type

Existing C code relies on the guarantee that `long` is the longest integral type. For example, the following commonly used idiom is necessary to print out an object of type `size_t`. Since `size_t` might be any unsigned integral type smaller than or equal to an `unsigned long`, the longest type must be assumed.

```
printf("%lu\n", (unsigned long)x);
```

If it suddenly becomes possible to map `size_t` to a larger type, such code will break when ported to systems that do so. It has been pointed out that no object can be statically declared to be large enough to be a problem. However, dynamically allocated objects whose sizes are derived from external data will still cause difficulties in this context.

This could be fixed by disallowing any mapping of the standard typedefs `size_t`, `ptrdiff_t`, and `sig_atomic_t` to `long long`. Even `wchar_t` is affected in the case that the user desires to print out the numeric value of a wide character.

The fact that on some machines `long long` may be the same size as `long` is irrelevant, since the purpose of `long long` is to create a type that can be larger than `long` on some systems (initially most of them).

## Promotion of Constants

On a 32-bit machine, the function call

```
f(-2147483648);
```

passes the most negative integer as an argument to the function. This author added a `long long` data type to a compiler for a company that used such calls in intentionally nonportable systems code that they were unwilling to change. Function prototypes were not used. When the promotion rules for integral constants were upgraded in the ''obvious'' way to accommodate `long long`, this code broke and the company mandated that constants never be promoted beyond the original C data types. Note that the same problem would occur with variable argument lists even in the presence of function prototypes.

For that company, this was fixed by requiring an `LL` suffix on all `long long` constants.

The problems listed so far can be overcome by demoting `long long` to a second-class type. The following problem has no such solution.

## Type Mapping Explosion

The more ways there are to map types to sizes, the more chances there are to create porting difficulties. To see just part of this problem, let us consider a restricted mapping space in which the only allowed sizes above `char` are 16, 32, 64, and 128 bits. Further, let us assume that no data type other than `long long` will ever map to 128 bits. For the traditional data types, this gives the following possible mappings for different machine types.

| short | int | long |
|---|---|---|
| 16 | 16 | 32 |
| 16 | 32 | 32 |
| 32 | 32 | 32 |
| 16 | 16 | 64 |
| 16 | 32 | 64 |
| 32 | 32 | 64 |
| 16 | 64 | 64 |
| 32 | 64 | 64 |
| 64 | 64 | 64 |

With `long long` added in, the number of possibilities doubles: the above mappings are repeated once for a `long long` size of 64 bits, and again for 128 bits. Even if some of the resulting combinations are unlikely, the increase in possible scenarios is significant. The `long long` type hinders portability rather than helping it.

## Conclusion

There is no doubt that `long long` exists in some implementations; this author created one of them. However, significant language design problems suggest that we seriously consider keeping the data type out of the C standard. It would have to be a second-class integral type to keep from breaking existing code, and this defeats some of the uses contemplated for it, for example, to widen `size_t` to correspond to 64-bit pointers. It would also cause C to have even more portability problems than it already has.