J16/04-0133
WG21/N1693
September 10, 2004
J. Stephen Adamczyk
Edison Design Group, Inc.
jsa@edg.com

Adding the `long long` type to C++ (Revision 1)

I propose that we add the `long long` integral type to C++. This is desirable to make C++ more compatible with C99 and with the draft Ecma TG5 C++/CLI standard.

Adding `long long` was proposed previously by Roland Hartinger in June of 1995, in J16/95-0115=WG21/N0715. At the time, `long long` had not been considered by the C committee, and the C++ committee was reluctant to add a fundamental type that was not also in C. Almost a decade later, the world looks different: `long long` is part of C99, and many major C++ compilers support it. It's time to standardize it in C++.

This is issue ES016 on the Evolution Working Group issues list.

This paper is a revised version of N1565. The revision is to complete the section on Working Draft changes.

What To Call It

Okay, let's deal with the "ick" factor first. Yes, "`long long`" is an ugly way of spelling a 64-bit integer type. Yes, it doesn't provide a growth path when we find we need a 128-bit integer type ("`long long long`", anyone?).

My advice: get over it. It's ugly, but it's standard (*de facto* for C++, and *de jure* for C). We can consider other possibilities in the future, but the only thing that makes sense now is `long long`.

The Uncontroversial Parts

If we can get past the spelling "`long long`", a number of aspects are relatively uncontroversial:

- There are two new integral types, `long long` and `unsigned long long`, which are at least 64 bits long.
- There are new suffixes for literal constants, `LL` and `ULL`, which indicate `long long` and `unsigned long long` constants, as in `9223372036854775807LL`.
- The usual arithmetic conversions and the integral promotion rules are updated to handle `long long` and `unsigned long long` operands.
- `enum`s and bit fields can have `long long` or `unsigned long long` type.
- The `printf` and `scanf` formatting strings have a new length specifier `ll`, which is used for

long long and unsigned long long arguments, as in %lld.
- <climits> has new macros LLONG_MIN, LLONG_MAX, and ULLONG_MAX.
- The C++ library has new overloads for long long and unsigned long long operands, e.g., for extractors and inserters.

Parts We Don't Need From C99

In C99, the long long extension is somewhat intertwined with two other changes:

1. Extended integer types, which allow implementations to have other sizes of integral types beyond those required by the standard (notably, bigger ones).
2. The <stdint.h> header, which provides names for types that map to specific sizes, e.g., int64_t for an exactly-64-bit signed integer. It also provides some useful types like intmax_t, which gives the largest available signed integer type.

I think these are fine, but also I think they can be considered separately from long long without problems down the road. They're not included in this proposal.

A Logistical Problem

The current C++ standard refers to the C library by reference, and it specifies the C89 version of the standard plus the 1995 amendment. I presume that the Library Working Group will be updating that. Ideally, changes like those for the printf and scanf formatting strings would be handled simply by pointing to the C99 standard. However, if that's not possible we could add text for specific additions to the C89 specifications.

The Controversial Part

C99 made one decision that's controversial. In the rules for determining the type of a literal constant, there are lists of types for various forms of constants. A constant's type is the first of the types on the list into which the value fits. That approach is the same in C99 and C++. However, in two cases the C99 lists are not simply what one would expect for a straightforward extension of the C89 and C++ lists:

1. For a decimal constant with no suffix, the C99 list is int, long int, long long int.
2. For a decimal constant with an l or L suffix, the C99 list is long int, long long int.

For upward compatibility with C89 and C++, unsigned long int should appear on each of those lists after long int[1]. It doesn't, and that means a constant with one of the forms above whose value is too large for long but fits in unsigned long, e.g., 4000000000 on an implementation with 32-bit longs, has a different type in C99 than it does in C89 or C++ (long long rather than unsigned long).

---

1. The C++ story is actually a little more complicated: probably for exactly this reason, the C++ standard says that if the value of an unsuffixed decimal constant does not fit in int or long int the behavior is undefined, which allows implementations to give an error or treat the constant as unsigned long.

The C committee made an explicit decision to introduce this incompatibility with C89. Why? It has to do with extended integer types and planning for the future. Consider a constant like 18446744073709551615, which is too big to fit in a 64-bit `long long`. If a C99 implementation has a nonstandard integer type larger than 64 bits, it would make sense to treat that constant as a signed value of that larger type. What if the implementation does not have such a type? Well, the constant would fit in `unsigned long long`, but allowing an implementation to use that type would be unfortunate, because the same constant would be signed on some implementations and unsigned on others. So to avoid that the C99 standard mandates an error for that constant if there is no signed type that can represent it. This gives consistent behavior across different implementations, and also allows the graceful addition of a future standard 128-bit integer type.

This is all fine and noble, and not a source of incompatibility, when applied at the upper end of the `long long` range, but the same principle is also applied at the upper end of the `long` range and there it makes a difference in a program like

```
#include <stdio.h>
int main() {
  if (4000000000 > -1) {
    printf(">\n");
  } else {
    printf("Not >\n");
  }
  return 0;
}
```

which produces different output in C99 than it did in C89. (C99 outputs “>”.)

Is this difference necessary? No. C99 could have left the types of those existing constants the same. (EDG’s front end does that when it supports `long long` in C++ mode currently.) But the consistent application of the rule does make sense: It guarantees that a constant that looks signed really is signed. That’s a good thing, because any use of an implicitly-unsigned constant is a potential bug. (In the above fragment, for example, it would be hard to argue that the C89/C++ behavior is more desirable than the C99 behavior.) If the programmer really does want an unsigned constant, he/she can request that explicitly by adding a `U` suffix to the constant, and that will improve the readability of his/her program and work the same way in C89 and C99.

So: C++ can either preserve upward-compatibility with C++98 or be compatible with C99. I recommend that we go with the C99 approach. In practice, there’s not much code that would be affected, and I’d bet that, in half the cases that change, the C99 behavior was what the programmer intended in the first place.

Detailed Working Draft Changes

In 2.13.1 [lex.icon], the syntax for integer constants needs to allow `ll` and `LL` suffixes as in C99 6.4.4.1. Mixed case (“`lL`”) is not allowed. If the suffix also includes a `U`, it can appear before or

after the `ll` or `LL`. Change the syntax as follows:

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
unsigned-suffix long-long-suffix
long-suffix unsigned-suffix$_{opt}$
long-long-suffix unsigned-suffix$_{opt}$
unsigned-suffix: one of
`u U`
long-suffix: one of
`l L`
***long-long-suffix:* one of**
`ll LL`

In 2.13.1 [lex.icon] p2, the rules for determining the type of a constant need to be extended to match the table in C99 6.4.4.1p5. Change 2.13.1 [lex.icon] p2 as follows:

The type of an integer literal depends on its form, value, and suffix. If it is decimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `long int`, **`long long int`**; if the value cannot be represented as a **`long`** `long int`, the behavior is undefined. If it is octal or hexadecimal and has no suffix, it has the first of these types in which its value can be represented: `int`, `unsigned int`, `long int`, `unsigned long int`, **`long long int`**, **`unsigned long long int`**. If it is suffixed by `u` or `U`, its type is the first of these types in which its value can be represented: `unsigned int`, `unsigned long int`, **`unsigned long long int`**. If it is **decimal and is** suffixed by `l` or `L`, its type is the first of these types in which its value can be represented: `long int`~~`, unsigned long int`~~**`, long long int`**. **If it is octal or hexadecimal and is suffixed by `l` or `L`, its type is the first of these types in which its value can be represented: `long int`, `unsigned long int`, `long long int`, `unsigned long long int`.** If it is suffixed by `ul`, `lu`, `uL`, `Lu`, `Ul`, `lU`, `UL`, or `LU`, its type is **the first of these types in which its value can be represented:** `unsigned long int`, **`unsigned long long int`**. **If it is decimal and is suffixed by `ll` or `LL`, its type is `long long int`. If it is octal or hexadecimal and is suffixed by `ll` or `LL`, its type is the first of these types in which its value can be represented: `long long int`, `unsigned long long int`. If it is suffixed by both `u` or `U` and `ll` or `LL`, its type is `unsigned long long int`.**

In 3.9.1 [basic.fundamental] p2 and p3, add `long long` and `unsigned long long`. See C99 6.2.5p4. Change 3.9.1 [basic.fundamental] p2 and p3 as follows:

> There are ~~four~~ **five** signed integer types : "`signed char`", "`short int`", "`int`", ~~and~~ "`long int`"**, and "`long long int`"**. In this list, each type provides at least as much storage as those preceding it in the list. Plain `int`s have the natural size suggested by the architecture of the execution environment; the other signed integer types are provided to meet special needs.
>
> For each of the signed integer types, there exists a corresponding (but different) unsigned inte-

> ger type: "`unsigned char`", "`unsigned short int`", "`unsigned int`", ~~and~~ "`unsigned long int`"**, and "`unsigned long long int`"**, each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. The range of nonnegative values of a signed integer type is a subrange of the corresponding unsigned integer type, and the value representation of each corresponding signed/unsigned type shall be the same.

Change footnote 44 on 3.9.1 [basic.fundamental] p7 that mentions `enum` integral promotion as follows:

> 44) Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to ~~`int`, `unsigned int`, `long`, or `unsigned long`,~~ **integral types** as specified in 4.5.

In 4.5 [conv.prom] p2, the integral promotions for `wchar_t` and `enum`s need to include `long long` and `unsigned long long`. Note that this paragraph deals with the promotion of those types to an integral type, not widening, so it does need to be changed. 4.5 [conv.prom] p3 on bit field promotions requires no changes. Change 4.5 [conv.prom] p2 as follows:

> An rvalue of type `wchar_t` (3.9.1) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: `int`, `unsigned int`, `long`, ~~or~~ `unsigned long`**, `long long int`, or `unsigned long long int`**. An rvalue of an enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of the enumeration (i.e., the values in the range b_{min} to b_{max} as described in 7.2): `int`, `unsigned int`, `long`, ~~or~~ `unsigned long`**, `long long int`, or `unsigned long long int`**.

In 5 [expr] p9, the usual arithmetic conversions rules, add after the fourth step:

- **Then, if either operand is `unsigned long long`, the other shall be converted to `unsigned long long`.**
- **Otherwise, if one operand is `long long` and the other `unsigned long int` or `unsigned int`, then if a `long long` can represent all the values of the unsigned operand type, the unsigned operand shall be converted to `long long`; otherwise both operands shall be converted to `unsigned long long`.**
- **Otherwise, if either operand is `long long`, the other shall be converted to `long long`.**

and change the "Then" at the start of the fifth step to "Otherwise". The C99 version of this is at 6.3.1.1 and 6.3.1.8; it uses a fancier approach involving assigning ranks to the integer types.

In 5.8 [expr.shift] p2 add the `unsigned long long` shift case, with reduction modulo `ULLONG_MAX+1`. The changes are as follows:

> The value of `E1 << E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bit positions; vacated bits are zero-filled. If `E1` has an unsigned type, the value of the result is `E1` multiplied by the quantity 2 raised to the power `E2`, reduced modulo **`ULLONG_MAX+1` if `E1` has type**

> **unsigned long long,** ULONG_MAX+1 if E1 has type unsigned long, UINT_MAX+1 otherwise. [*Note:* the constants **ULLONG_MAX,** ULONG_MAX**,** and UINT_MAX are defined in the header <climits>. --*end note*]

In 7.1.5.2 [dcl.type.simple] add the new type specifier combinations long long, signed long long, long long int, signed long long int, unsigned long long, and unsigned long long int in Table 7.

In 7.2 [dcl.enum] p5, no change is needed for the underlying type of an enum.

In 16.1 [cpp.cond] p4, int, unsigned int, long, and unsigned long preprocessing expressions should be remapped to long long and unsigned long long. See C99 6.10.1p3, which uses intmax_t and uintmax_t. Change a sentence in 16.1 [cpp.cond] p4 as follows:

> The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.19 using arithmetic that has at least the ranges specified in 18.2, except that ~~int and unsigned int~~ **all signed and unsigned integer types** act as if they have the same representation as, respectively, **long** long and unsigned **long** long.

The library sections need to be updated. This is handled by the library TR1 changes. An earlier version of the required changes was in N1568 by P.J. Plauger. Note that apparently the only thing that establishes minimum sizes for the integral types in the C++ standard is the requirement that they match the <climits> macro values like INT_MAX; see the footnote in 3.9.1 [basic.fundamental] p2. Make this explicit by deleting the footnote

> ~~40) that is, large enough to contain any value in the range of INT_MIN and INT_MAX, as defined in the header <climits>.~~

and adding at the end of 3.9.1 [basic.fundamental] p3:

> **The signed and unsigned integral types shall be large enough to contain the maximum and minimum values for those types defined by the macros in <climits> (18.2.2 [lib.c.limits]). [*Example:* int must be large enough to contain the INT_MIN and INT_MAX values. --*end example*]**