J16/05-0006
WG21/N1746
2004-12-29
J. Stephen Adamczyk
Edison Design Group, Inc.
`jsa@edg.com`

Adding extended integer types to C++

I propose that we add extended integer types to C++. This is desirable to make C++ more compatible with C99 and with the draft Ecma TG5 C++/CLI standard.

Extended integer types are simply implementation-specific integer types provided in addition to the standard integer types. They could be bigger than the largest standard type, or have a size between two standard types. An implementation on an architecture that has 128-bit integers, for example, could provide an extended integer type that maps to those.

This proposal is part of a set of three related proposals to bring C99 features into C++: first, adding `long long`, which was covered by my paper N1735; second, adding extended integer types, which is the subject of the present paper; and third, adding the `<stdint.h>` header, which is included in the library group TR1.

Extended integer types are a funny feature, in that there's no way to use them in a portable program. The type specifier representations of such types aren't standardized (they're extensions, after all), so there's no standard way to name them. So what good is the extension? It provides a framework that does two things:

- It requires that if implementations add additional integer types they do so in conformance with certain rules.
- It guarantees that the behavior of standard-conforming programs will not be affected by the presence of extended integer types in an implementation.

A final point on implementation cost: this extension will probably cause no changes in most compilers. Any compiler that has no integer types other than those mandated by the standard (and some version of `long long`, which is mandated by the N1735 change) will likely conform already.

Detailed Working Draft Changes

Note: in most cases, these changes hit the same places modified by N1735. The "before" wording shown here is the wording after application of the changes from that paper.

3.9.1 [basic.fundamental] paragraphs 2 and 3 need to be changed as follows to define signed and unsigned extended integer types and to adjust the definition of signed and unsigned integer types to include the extended versions. See C99 6.2.5p4, p6, and p7.

> There are five ***standard*** *signed integer types* : "`signed char`", "`short int`", "`int`", "`long int`", and "`long long int`". In this list, each type provides at least as much storage as those preceding it in the list. **There may also be implementation-defined *extended signed integer types*. The standard and extended signed integer types are collectively called *signed integer types*.** 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 **standard** signed integer types, there exists a corresponding (but different) ***standard*** *unsigned integer type*: "`unsigned char`", "`unsigned short int`", "`unsigned int`", "`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. **Likewise, for each of the extended signed integer types there exists a corresponding *extended unsigned integer type* with the same amount of storage and alignment requirements. The standard and extended unsigned integer types are collectively called *unsigned integer types*.** 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. **The standard signed integer types, standard unsigned integer types, and the `bool` type are collectively called the *standard integer types*, and the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.**

2.13.1 [lex.icon] paragraph 3 needs to be changed as follows to consider extended integer types for integer literals whose values do not fit in the standard integer types. See C99 6.4.4.1p5. Note that there are no suffixes indicating extended integer types, and an extended integer type smaller than `long long` will never be used as the type of an integer literal.

> **If an integer constant cannot be represented by any type in its list, it may have an extended integer type, if the extended integer type can represent its value. If all of the types in the list for the constant are signed, the extended integer type shall be signed. If all of the types in the list for the constant are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned.** A program is ill-formed if one of its translation units contains an integer literal that cannot be represented by any of the allowed types.

16.1 [cpp.cond] paragraph 4 needs to be changed as follows to indicate that preprocessing expressions should be evaluated in the largest available integer types. See C99 6.10.1p3, which uses `intmax_t` and `uintmax_t`.

> 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 all signed and unsigned integer types act as if they have the same representation as, respectively, ~~`long long int` and `unsigned long long int`~~ **the largest signed integer type or unsigned integer type**.

4.5 [conv.prom] paragraph 1 needs to be changed as follows to define integral promotions for small extended integer types. This uses the concept of “rank” introduced below. See C99 6.3.1.1p2.

> An rvalue of ~~type `char`, `signed char`, `unsigned char`, `short int`, or `unsigned short int`~~ **an integer type whose integer conversion rank (4.13 conv.rank) is less than the rank of `int`** can be converted to an rvalue of type `int` if `int` can represent all the values of the source type; otherwise, the source rvalue can be converted to an rvalue of type `unsigned int`.

Add a new section 4.13 [conv.rank] as follows to provide the definition of “integer conversion rank.” See C99 6.3.1.1. (The text below is the C99 wording almost unchanged.)

> **4.13 Integer conversion rank**
>
> **Every integer type has an *integer conversion rank* defined as follows:**
> - **No two signed integer types shall have the same rank, even if they have the same representation.**
> - **The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller size.**
> - **The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.**
> - **The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.**
> - **The rank of any standard integer type shall be greater than the rank of any extended integer type with the same size.**
> - **The rank of `char` shall equal the rank of `signed char` and `unsigned char`.**
> - **The rank of `bool` shall be less than the rank of all other standard integer types.**
> - **The rank of any enumerated type shall equal the rank of its underlying type (7.2 dcl.enum).**
> - **The rank of any extended signed integer type relative to another extended signed integer type with the same size is implementation-defined, but still subject to the other rules for determining the integer conversion rank.**
> - **For all integer types T1, T2, and T3, if T1 has greater rank than T2 and T2 has greater rank than T3, then T1 has greater rank than T3.**
>
> **[*Note*: The integer conversion rank is used in the definition of the integral promotions (4.5 conv.prom) and the usual arithmetic conversions (5 expr). -- *end note*]**

In 5 [expr] paragraph 9, the text from the fourth bullet on is changed from

> - Otherwise, the integral promotions (4.5) shall be performed on both operands.
> - ~~Then, if either operand is `unsigned long long int`, the other shall be converted to `unsigned long long int`.~~
> - ~~Otherwise, if one operand is `long long int` and the other `unsigned long int` or `unsigned int`, then if a `long long int` can represent all the values of the unsigned~~

~~operand type, the unsigned operand shall be converted to `long long int`; otherwise both operands shall be converted to `unsigned long long int`.~~

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

~~[*Note*: otherwise, the only remaining case is that both operands are int -- *end note*]~~

to (see C99 6.3.1.8):

- Otherwise, the integral promotions (4.5) shall be performed on both operands. **Then the following rules are applied to the promoted operands:**
- **If both operands have the same type, then no further conversion is needed.**
- **Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.**
- **Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.**
- **Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.**
- **Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.**