Standardizing Extended Integers

Document number: P0989R0
Date: 2018-04-01
Audience: EWG
Reply-to: Tony Van Eerd. extint at forecode.com

Along the lines of JF Bastien's "Signed Integers are Two's Complement" (p0907) we recommend standardizing the *extended integer types* (ie __int128_t et al, but with better names), as another example of standardizing existing practice, and acknowledging existing and upcoming hardware. (Note, we are not *requiring* 128 bit integers, but standardizing the naming of extended integers.) Along the way, we will also define *a simpler, more consistent, yet more powerful way to declare integers of different sizes*.

Acknowledgements

This work is based on initial conversations with Dr. Bjarne Stroustrup.

Motivations

1. __int128_t is becoming more and more common - and more essential to everyday coding.
2. There will probably be other sizes in the future.
3. *Portability*: Integer sizes are not portable. Not even the *relative* sizes between the integer types. How can I know whether to type `int` or `long`?
4. UB. It is hard to know the size of integers such as `long int` etc, particularly relative to each other. For example, to be safe from overflow (UB) when multiplying `int` x `int`, you require an integer type with `size >= 2 * sizeof(int)` to safely hold the result. But short of asserts, it is hard to specify/guarantee the right type. In short, users find the `=` of `sizeof(long) >= sizeof(short)` surprising and confusing.
5. Use of `long`, `short`, `long long`, etc is confusing and inconsistent - simpler rules (less special cases!) would make simple things simple.
6. Bit fields are esoteric and don't follow the same rules as other types (eg structured bindings, initialization ambiguity) - using the same notation for replacing __int128_t, we can also fix bit fields. Again, let's make simple things simple.

Short Story

We propose that the `long` and `short` modifiers be allowed to be repeated, and combined, in the obvious way.

Long Story

Note that currently, although platform specific, it is common for the `short` and `long` modifiers to exactly double and half (irrespectively) the number of bits in an int:

```
short int x;      // = 32/2   = 16 bits
int y;            // = 32*1   = 32 bits
long int z;       // = 32*2   = 64 bits
long long int w; // = 32*2*2 = 128 bits
```

Imagine a system that did follow a half/double rule - then a user might naturally see this as a pattern and logically expect the following to also work:

```
long long long int v;  // = 32*2*2*2 = 256 bits
```

Unfortunately, this gives us the famous gcc error msg:

```
    prog.cc:15:11: error: 'long long long' is too long for GCC
     long long long v;
               ^~~~
```

(Note this error is so common that they have a dedicated msg for it. This implies many things: common user need, request, confusion, and *expectation*.)

1. Consistently name larger sizes

Thus we propose that the language allow multiple applications of `long`, as many as the platform supports. Instead of implementation specific names/hacks like `__int128_t`, use `long long int`, etc.

2. Consistent doubling

To be clear, not only are we recommending that `long long long int` be supported (on platforms that have that size), but that it *always* be twice as long as `long long int`. ie each additional `long` *always* doubles the size of what would have otherwise been declared. Note: this includes `long int`. It should *always* be twice as long as `int`. (We can grandfather in, but deprecate any systems where this is not already currently the case.)

3. Consistent halving

As is alread common:

```
short int u; //  = 32/2 = 16 bits
```

We propose that `short` *always* halves. (Again, we can allow a deprecation period for odd systems.)

Additionally, to be consistent and match users' expectations, we propose allowing repetition of `short`:

```
short short int b; // = 32/2/2 = 8 bits, ie std::byte
```

4. Combining

Users will likely expect to be able to combine `short` and `long` (particularly in templatized code).

There may be some ambiguity in expectations here, so we need to be careful:

```
short long int i; // = 32*2/2 = 32 bits, ie same as int ?
```

Some users may expect that this would first double the bits (applying `long`), then half them (applying `short`).

This is obviously useless. To make it useful, we should interpret it slightly differently - and in a way that is actually *more intuitive*: a `short long int` is a long `int`, but not *too* long, an integer still longer than `int`, but shorter than a typical `long int`. ie a short `long int`, (which *is* what the user typed).

Modifier Modify Modifiers

This can be understood by seeing that the `short` modifier modifies the `long` modifier (ie apply left to right instead of right to left). ie a "short-long int", not a "short long-int". ie when someone wants a longer `int`, but not *that* long, just "longish" or "kinda almost long but not quite" - in fact it is half long-ish, since `short` halves the `long`-ness modifier.

So since `long` adds 100% to the size of the `int`, `short long` would only add 50% to the size of the `int`. ie:

```
short long int j; // (1 + 1/0.5)(32) = 48 bits
```

But what if we reverse `short long` to `long short`? Doesn't the `long`, modifying the `short`, still result in cancellation? (or does the long *double* the effect of the shortening? ie making a `long short int` 8 bits?)

In one word, *No*. In two, *Obviously Not*.

If `int` is 32 bits, then a `long short int` is 24 bits.

The exact rule is thus obvious, and doesn't need explaining. But note the consistency:

- `short long` - short *takes away half* the bits you would have gained with `long`;
- `long short` - long *gains back half* the bits you would have lost with `short`.

Thus instead of `long` and `short` annihilating each other completely, they only cancel half the affect.

Further combinations are applied naturally:

```
short short long int; // 40 bits
```

`short short` can be thought of as "very short", a pattern for building words ("reduplication") which is already true in many human languages such as Swahili.

More examples:

```
short int -> 16 bits
long short int -> 24 bits
long long short int -> 28 bits
long long long short int -> 30 bits
```

To reach the bit lengths in between, we can interleave long and short as necessary:

```
short int -> 16 bits
long short int -> 24 bits
short long short int -> 20 bits
long short long short int -> 22 bits
short long short long short int -> 21 bits
```

A `short long short int` is thus a "kinda short longish short int"

Like a binary search, or Zeno's paradox, each step covers half the distance of the previous step - in the same direction if the modifier is same as the previous, or in the opposite direction if switching between modifiers. For this reason, we sometimes refer to these as Zeno Integers.

```
short long long long short int -> 29 bits
```

Think of this as:

```
short int -> 16bits - hmm, too short
long short int -> 24bits - getting there, needs to be longer
long long short int -> 28 bits - close, just a bit more
long long long short int -> 30bits - too far!
short long long long short int -> 29bits - just right!
```

(This is intuitive, but to be pedantically clear: note that the initial modifiers double (or halve) and then the 'Zeno walking' only applies once modifiers are mixed.)

Bit fields

We can now see that bit fields fall out of this naturally:

C++17	C++20

```
{
    int b12 : 12;
    int b5  : 5;
}
```

```
{
    long short short int b12;
    short long short short short int b5;
}
```

Concerns, Consequences

Attentive readers may have wondered what to do with something like:

```
short short short short short short int h; // 0.5 bits??
```

For a platform with `int` = 32bits, applying repeated halving would result in an `int` of size 0.5bits. (and of course, proper mixing or `short` and `long` could conceivably result in sizes such as 13.25 bits, etc)

Isn't this a problem? Particularly with portability?

No, not really. For half (or smaller) bits, we can use what the authors like to call *bit entanglement*.

Note that it takes 1 bit (or two halfbits) to have enough memory to differentiate between a 1 and a 0. (it takes 0 bits to store a `0` or 0 bits to store a `1` - see `std::true_type` and `std::false_type`, but takes a full bit (two halfbits) to store a value that could be either)

So a single half-bit needs to share state with another half-bit elsewhere in the program. When a half-bit is flipped between `0` and `1`, the corresponding entangled half-bit must also flip. For quarter-bits, all three other quarter-bits flip, etc.

It can thus be shown that 16 `short short short short short short int`s (ie 16 half-bits on many platforms) hold the same amount of information/entropy as a single `short short int` (ie 8bits)

Implementation Flexibility

For many platforms and implementations, it may be simplest to entangle *contiguous* fractional bits, but the authors feel that this may be both a burden and a pessimization in some cases, thus which fractional bits are entangled with which others should remain *implementation defined*.

Pragma pack

There may be cases where the total size of fractional bits does not add to a whole number. It is expected that many implementations will simply round up, "wasting" fractional bits. While this may be undesirable on embedded systems, etc, we believe this is an acceptable burden. It is expected that implementations will use `pragmas` such as `pragma pack(0.5)` to help users guide their compilers.

Alternative Entanglement

Alternatively, to avoid waste, an implementation could share entanglements *across systems*, eg between programs running on the same OS, or across the internet, etc. Alternatively, it would be possible to entangle partial bits with other partial bits at different points in time, instead of space. An entangled bit could, in fact, to save space, entangle with its future self.

Also, as may be obvious, newer systems (for example, quantum computers?) not directly tied to the notion of whole bits, could use other techniques, such as entangled particles, etc, to implement entangled bits. In fact the authors anticipate and encourage this direction, and is one of the reasons for choosing the term "bit entanglement".

Floating point types

See also P0192 (short float)

Short story: obviously the exact same long/short rules can and should also apply to floats and doubles.

Diacritics

We also propose allowing the use of diacritics (on the `o`) for `long` and `short`. ie:

`lōng int x;` == `long long int x;`

`shōrt int x;` == `long short int x;`

`löng int x;` == `short long int x;`

etc. Repeated applications of the diacritics would do the obvious thing:

`lō̈ng int x;` == `long short long int x;`

The diacritic(s) would also be allowed on the `i` of `int`.

`ïnt x;` == `short int x;`

Future Directions

Entangled Classes

C++ strives to ensure user defined classes enjoy the same language support as built in types. A future revision of this paper may also extend the `long` and `short` modifiers to user defined classes. In particular, *entangled classes* may be an elegant language-based notation for the *common* coding pattern of "action at a distance" wherein changing one object over *here* causes a change to another object over *there* arbitrarily far away (in 'code-distance' measured by lines of code, translation units, etc). This is a pattern found in almost all code bases, but it currently lacks direct language support - possibly in any language, not just C++.

bool

using bool = short short short short short int; means bool can now be 1 bit. vector<bool> is finally fixed. (Also, this allows a convenient short hand for half-bits: short bool - note that a recent paper by Sutton and Liber suggested a short bool would hold no information. This is incorrect; it obviously can hold half as much information as a bool.)

signed, unsigned

It is somewhat obvious that signed and unsigned could have similar reduplication rules, to allow for multi-dimensional numbers (ie signed signed could be "positive", "negative" or 2 other directions, either "very positive"/"very negative" or "left"/"right" or "up"/"down" etc. A unsigned signed signed int would have 6 signs, say "up/down/top/bottom/charm/strange" and could be directly applied to quarks. An unsigned signed bool would be a built in implementation of a tri-bool type. Unused sign bits can, of course, be entangled on systems where bits are at a premium.

lock-free programming

Entangled fractional bits would have obvious benefits in lock-free programming. It is assumed/implied in this proposal that std::atomic will work (lock-free) with fractional bits.

More Acknowledgements

Additional refinement via conversations with co-workers at Christie Digital, including Norm Ross and Colin Yardley.

See Also

http://wg21.link/p0192

http://wg21.link/p0907

http://2g21.link/p0999

https://en.wikipedia.org/wiki/Reduplication

PXXXXR0 "A Unit Type for C++", Sutton and Liber. via email.