std::int_least128_t
1. Introduction
2. Motivation
3. Impact on the standard
4. Impact on implementations
5. Design
1. Introduction

Proposed types for `<cmath>`

```cpp
std::int_least128_t    std::int_fast128_t
std::uint_least128_t   std::uint_fast128_t
```

Desired semantics

- Width of ≥ 128 bits (minimum-width types)
  - `std::int128_t` and `std::uint128_t` by proxy (exact-width types)
- Types are mandatory
- Types are extended integer types
- Strong standard library support
  - Some library changes required
2. Motivation

128-bit integers are useful

- Code search `/int128|int_128/ language:c++` → 145K files
  - For reference, `/std::byte/ language:c++` → 45.6K files
  - For reference, `/long double/ language:c++` → 582K files
- Used in many domains:
  - Cryptography and random number generation
  - Widening, multi-precision, fixed-point arithmetic
  - Implementing, parsing, printing (decimal) floating-point
  - Huge numbers (high-precision time, financial systems, etc.)
  - UUID, IPv6
  - Bitsets, bit-manipulation
  - ...
## 2. Motivation

### The push for 128-bit integers

<table>
<thead>
<tr>
<th>Language</th>
<th>Support/Evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>__int128, _Signed128, _BitInt(128)</td>
</tr>
<tr>
<td>C</td>
<td>_BitInt(128)</td>
</tr>
<tr>
<td>CUDA</td>
<td>__int128</td>
</tr>
<tr>
<td>C#</td>
<td>Int128</td>
</tr>
<tr>
<td>Rust</td>
<td>i128 (RFC-1504)</td>
</tr>
<tr>
<td>Swift</td>
<td>SE-0425</td>
</tr>
<tr>
<td>Go</td>
<td>golang/go/issues/9455</td>
</tr>
</tbody>
</table>

Many languages also support 128-bit through **multi-precision integers** in the standard library.
2. Motivation

128-bit integers have hardware support

<table>
<thead>
<tr>
<th>Operation</th>
<th>x86_64</th>
<th>ARM</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 → 128-bit unsigned multiply</td>
<td>mul</td>
<td>umulh, mul</td>
<td>mulhu, mul</td>
</tr>
<tr>
<td>64 → 128-bit signed multiply</td>
<td>imul</td>
<td>smulh, mul</td>
<td>mulsu, mul</td>
</tr>
<tr>
<td>128 → 64-bit unsigned divide</td>
<td>div</td>
<td>N/A</td>
<td>divu (RV128I)</td>
</tr>
<tr>
<td>128 → 64-bit signed divide</td>
<td>idiv</td>
<td>N/A</td>
<td>divs (RV128I)</td>
</tr>
<tr>
<td>64 → 128-bit carry-less multiply</td>
<td>pclmulqdq</td>
<td>pmull, pmull2</td>
<td>clmul, clmulh</td>
</tr>
</tbody>
</table>
2. Motivation

Motivating example
Using 128-bit integers, `isinf(float128_t)` can be implemented as follows:

```cpp
constexpr float128_t abs(float128_t x) {
    return bit_cast<float128_t>(
        bit_cast<uint128_t>(x) & (uint128_t(-1) >> 1));
}

constexpr bool isinf(float128_t x) {
    return bit_cast<uint128_t>(abs(x))
        == 0x7fff'0000'0000'0000'0000'0000'0000'0000;
}
```
3. Impact on the standard

C Compatibility

• ABI issues related to `intmax_t` have been resolved in C23.
• `std::int_least128_t` does not imply existence of `int_least128_t` in C.
• `std::printf` support for 128-bit must be optional.

Core language impact
None. (extended integer semantics are just fine)

Standard library impact

• Menial changes (adding macros, aliases, etc.)
• Enhancing support for extended integers (`std::to_string`, `std::bitset`, etc.)
• Preventing 128-bit integers from breaking ABI (`std::ranges::iota_view`)
3. Impact on the standard

Enhancing support for extended integers

- Some overload sets (std::abs, std::to_string, std::bitset constructor) are restricted to standard integer types.
- Adding overloads for std::int_least128_t would not comply.

```cpp
// current overload set
constexpr int abs(int j);
constexpr long int abs(long int j);
constexpr long long int abs(long long int j);

// proposed overload set
constexpr signed-integer-least-int abs(signed-integer-least-int j);
```
4. Impact on implementations

Implementing `std::int_least128_t`
- GCC and clang provide `_BitInt(128)` and `__int128` (with some restrictions).
- No built-in type for MSVC, only `std::_Signed128, std::_Unsigned128` classes.

Implementing standard library (non-)changes
- Many menial changes (defining macros, aliases, relaxing constraints, ...)
- Numerics and bit manipulation (`std::gcd, std::popcount, ...`)
- New overloads (`std::abs, std::to_string, std::bitset`)
- 256-bit arithmetic for `std::linear_congruential_engine<std::uint128_t>`
- Overwhelming majority of standard library unchanged.
- As mentioned before, 128-bit `std::printf` support is optional.
5. Design

Questions

• “Why no standard integer?“
  • long long long is too long; also, this would break ABI.
• “Why no std::int_least256_t?“
  • Too little motivation, unclear ABI, long literals.
• “Why not solve this more generally (e.g. _BitInt(N))?“
  • Huge effort, better done through std::big_int<N>.
• “Why make it mandatory?“
  • If it’s optional, library authors do twice the work.
  • Implementation effort is reasonable, software emulation acceptable.
  • It’s already here: _BitInt(128), __int128, std::_Signed128.
• “Why rely on extended integer semantics?“
  • No core wording changes; semantics are desirable.
Jan Schultke; P3140: std::int_least_128_t (latest revision)