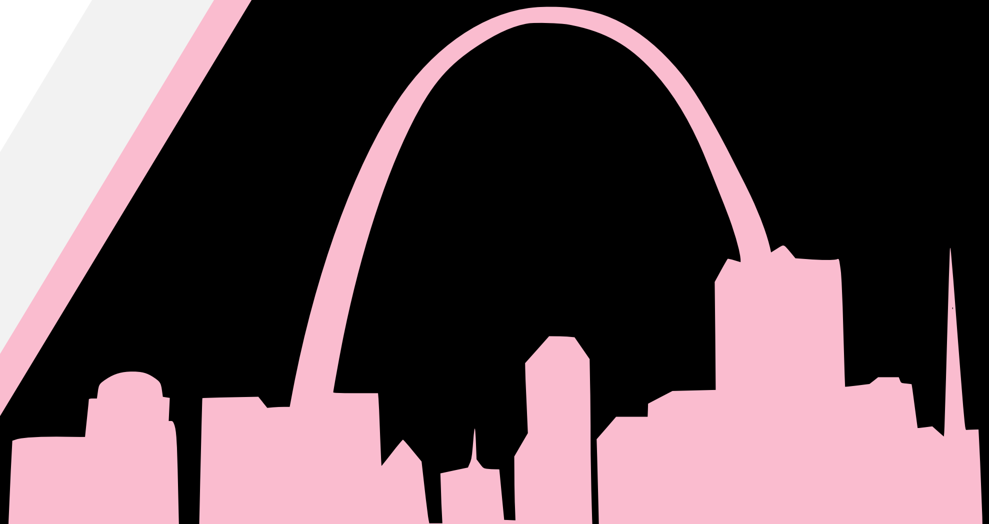
P3140R0

`std::int_least128_t`

St. Louis
2024



Author: Jan Schultke
Presenter: Jan Schultke

Audience: EWG
Project: ISO/IEC 14882 Programming Languages — C++, ISO/IEC JTC1/SC22/WG21

Contents

1. Introduction

Proposed types

- 128-bit signed and unsigned integers
- `std::int_least128_t`, `std::uint_least128_t`, etc. spellings

Desired semantics

- Width of ≥ 128 bits (minimum-width types)
    - `std::int128_t` and `std::uint128_t` by proxy (exact-width types)
- Types are mandatory (at least on hosted platforms)
- Types are extended integer types (currently proposed, subject to change?)
- 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

Language	Support/Evolution
C++	`__int128`, `_Signed128`, `_BitInt(128)`
C	`_BitInt(128)`
CUDA	`__int128`
C#	`Int128`
Rust	`i128` (RFC-1504)
Swift	SE-0425
Go	golang/go/issues/9455

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

2. Motivation

128-bit integers have hardware support

Operation	x86_64	ARM	RISC-V
64 → 128-bit unsigned multiply	`mul`	`umulh, mul`	`mulhu, mul`
64 → 128-bit signed multiply	`imul`	`smulh, mul`	`mulsu, mul`
128 → 64-bit unsigned divide	`div`	N/A	`divu (RV128I)`
128 → 64-bit signed divide	`idiv`	N/A	`divs (RV128I)`
64 → 128-bit carry-less multiply	`pclmulqdq`	`pmull, pmull2`	`clmul, clmulh`

2. Motivation

Motivating example

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

```
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

Depends on the design (no changes in the current proposal)

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.

```
// 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

- *“What if we added a standard integer instead?“*
  - long long long?
  - More core impact.
  - Difference in conversion rank.
- *“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.

6. Open questions

- *“Do we want 128-bit fundamental type integers in the language? “*
  - If not, drastic changes to proposal needed
- *“Is a non-general solution (no 256-bit, no arbitrary bit, etc.) worth pursuing?“*
- *“Do we want them to be (mandatory) extended integers, or standard integers?“*
  - Observable difference: conversion rank vs. `long long`
- *“Should 128-bit integers be mandatory for freestanding implementations?“*
  - Author position: neutral
- *“Should 128-bit integers be entirely optional? “*
  - Author position: strongly against
- *“Is it acceptable for them to be a distinct type from `__int128`, even on implementations that already support such a 128-bit integer type?“*
- *“Do we pursue library changes which increase support for additional integer types (extended or otherwise) and prevent them from breaking ABI? “*

References

Jan Schultke; **P3140:** `std::int_least_128_t` (latest revision)
https://eisenwave.github.io/cpp-proposals/int-least128.html