Boost.SafeNumbers: A Proposal

Table of Contents

Introduction . . . 1
Boost.SafeNumbers . . . 1
Goals . . . 2
- Provide Concrete Types . . . 2
- Provide Options for Runtime Behavior . . . 2
- Complete Standard Library Support . . . 3
- Acceptable Runtime Penalty . . . 3
- Human-Readable Error Messages . . . 3

Analysis of Alternatives . . . 3
- Boost.Safe_Numerics . . . 3
- Intel Safe Arithmetic . . . 5
- Existing Compiler Flags and Sanitizers . . . 5

Conclusion . . . 5
Sources . . . 5

Introduction

Much recent effort in improving the safety of C++ has revolved around memory safety and library hardening [1][2][4]. These are incredibly important undertakings in light of the US Government's recommendations to move to memory safe languages [5]. Another source of errors in C++ come from the behavior of arithmetic. It is legal to compare a signed integer to an unsigned integer yet the resulting behavior can be unexpected. Take for example: -99 < 340282366920938463463374607431768211408. This should obviously evaluate to `true`, but it is in fact `false` because these values are bitwise equal (as unsigned 128-bit integers). The goal is to avoid this situation entirely by offering compile-time as well as runtime type safety guarantees much like the primitive types in Rust [10] do.

Boost.SafeNumbers

> I do not want to see any longer a bool became an int.
>
> — u/RoyBellingan[3]

The aim of Boost.SafeNumbers is to be used as a complete replacement for the builtin numeric types of C++. We have several goals that align with this desire.

Goals

Provide Concrete Types

The concrete types the library will provide are:

1. Unsigned Integers: u8, u16, u32, u64, and u128
2. Signed Integers: i8, i16, i32, i64, and i128
3. Floating Point: f32 and f64

Each of these types will have the following properties:

1. No implicit conversions (to include to boolean values)
2. No mixed sign operations
3. No operators defined between Boost.SafeNumbers and builtin arithmetic types except for explicit construction
4. Any numerical error such as overflow will `throw` on error by default at runtime, and fail at compile time when possible.

These types offer alternatives to all the standard numerical types in C++ as well as the often requested 128-bit integer types.

Provide Options for Runtime Behavior

The first option is to allow for `-fno-exceptions` environments. To do this we will internally use `boost.throw_exception` which already has well-defined results depending on the mode of operations

Second the types will have a number of different operators much like they do in Rust[8]:

```
class u32
{
    // Throws/terminates on overflow
    friend constexpr u32 operator+(u32 lhs, u32 rhs);

    // Saturates to std::numeric_limits<u32>::max()
    friend constexpr u32 saturating_add(u32 lhs, u32 rhs) noexcept;

    // Allows well defined wrapping behavior
    friend constexpr u32 wrapping_add(u32 lhs, u32 rhs) noexcept;

    ...
};
```

Complete Standard Library Support

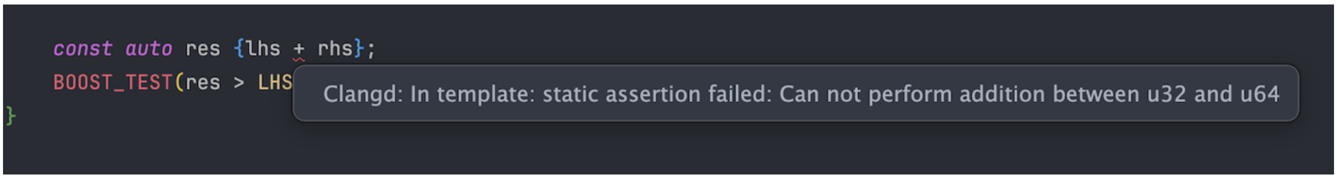
Since our goal is to provide a complete replacement for the built-in numeric types we need to provide a complete standard library including support for: `<charconv>`, `<format>`, `<iostream>`, `<cmath>`, {fmt}, etc.

Acceptable Runtime Penalty

Experimentation with Boost.Unordered with shows that Fil-C come with an ~2.5x runtime [9]. By leveraging existing compiler intrinsics to detect overflow (e.g. `__builtin_add_overflow`) the goal is to have a runtime penalty of less than 2x. If the runtime penalty is too great the additional safety offered by this library may not be sufficiently convincing for use.

Human-Readable Error Messages

In many cases such as mixed type operations we intend to emit a compiler error. To make these compiler errors useful they must be trivially human-readable.



```
    const auto res {lhs + rhs};
    BOOST_TEST(res > LHS
}
```

Clangd: In template: static assertion failed: Can not perform addition between u32 and u64

Figure 1. An example of an error from mixed type operations that is easily read through ones IDE

Analysis of Alternatives

Boost.Safe_Numerics

Boost.safe_numerics [7] offers a users the ability to wrap types before an operation at runtime, with policy based results on error.

Example 1. This example on how to use Boost.safe_numerics is directly from the library documentation

```
try
{
    using namespace boost::safe_numerics;
    safe<std::int8_t> x = INT_MAX;
    safe<std::int8_t> y = 2;
    safe<std::int8_t> z;
    // rather than producing an invalid result an exception is thrown
    z = x + y;
}
catch(const std::exception & e)
{
    // which we can catch here
    std::cout << "error detected:" << e.what() << std::endl;
}
```

This example when using our proposal would instead look something like:

Example 2. An annotated version of the Example 1 except using Boost.SafeNumbers instead

```
try
{
    using namespace boost::safe_numbers;
    // u8 x = INT_MAX; // Compile error: implicit narrowing is not allowed

    const u8 x {UINT8_MAX};
    const u8 y {2};
    const u8 z {x + y};

    std::cout << "Value of z: " << z << std::endl;
}
catch(const std::exception & e)
{
    std::cerr << "Error Detected: " << e.what() << std::endl;
}
```

Expected Output

```
Error Detected: Overflow detected in unsigned addition
```

By making the values constexpr in the above example we can move the errors from run time to compile time. This should lead to fewer instances where exceptions are possible.

Example 3. Our safe_numbers examples from above but with constexpr evaluation of addition instead of runtime

```
try
{
    using namespace boost::safe_numbers;

    constexpr u8 x {UINT8_MAX};
    constexpr u8 y {2};
    constexpr u8 z {x + y};

    std::cout << "Value of z: " << z << std::endl;
}
catch(const std::exception & e)
{
    std::cerr << "Error Detected: " << e.what() << std::endl;
}
```

Expected Output

```
error: constexpr variable 'z' must be initialized by a constant expression
   19 |         constexpr u8 z {x + y};
      |                      ^ ~~~~~~~

note: non-literal type 'std::overflow_error' cannot be used in a constant
expression
  156 |         BOOST_THROW_EXCEPTION(std::overflow_error("Overflow detected in
unsigned addition"));
```

Intel Safe Arithmetic

Boost.SafeNumbers offers to provide a smaller set of features (e.g. no arbitrary precision arithmetic) with greater interoperability with the STL. This library [11] is also in active development.

Existing Compiler Flags and Sanitizers

GCC, Clang, and other offer "-Wsign-conversion", "-Wconversion", and "-Wfloat-equal" can be used today to avoid large classes of bugs, and we highly recommend their use. These flags along with UBSan will catch large classes of bugs. Boost.SafeNumbers aims to take this a step further in rigidity of rules. For example, UBSan will allow rollover of an unsigned integer, whereas by default Boost.SafeNumbers will throw.

Conclusion

In conclusion, Boost.SafeNumbers aims to offer a library level solution to issues of arithmetic safety in C++. By offering solid performance, an expansive standard library, and readable compiler errors it should be viewed as an acceptable alternative to the built-in types and STL.

Sources

1. https://spawn-queue.acm.org/doi/pdf/10.1145/3773097
2. https://isocpp.org/files/papers/P3471R1.html
3. https://www.reddit.com/r/cpp/comments/1oqd01r/what_do_you_dislike_the_most_about_current_c/
4. https://fil-c.org
5. https://www.cisa.gov/resources-tools/resources/product-security-bad-practices
6. https://en.cppreference.com/w/cpp/header/stdfloat.html

7. https://github.com/boostorg/safe_numerics/tree/develop
8. https://doc.rust-lang.org/std/primitive.u32.html
9. https://bannalia.blogspot.com/2025/11/some-experiments-with-boostunordered-on.html
10. https://doc.rust-lang.org/std/#primitives
11. https://github.com/intel/safe-arithmetic