Low-Level Integer Arithmetic

Document #:	P3018R0
Date:	2023-10-15
Project:	Programming Language C++ SG6 Numerics Library Evolution Group

Reply-to:	Andreas Weis <cpp@andreas-weis.net>

Abstract

This paper proposes a number of library functions for performing basic integer arithmetic. Unlike the built-in language facilities, these operations can not trigger undefined behavior and closely resemble the operations provided by hardware. They are intended to be used as building blocks by library writers working on integer-heavy libraries.

Contents

1 Introduction

The built-in integer operations are among the oldest parts of C++, carried over virtually unchanged since the inception of C in the early 1970s. The requirements for integer operations were very different back then, with one of the main challenges being the task of finding a common abstraction for the many different machine architectures and their various forms of handling integer operations and representations.

The world of integer arithmetic today is significantly more homogeneous. For C++20, WG21 has agreed that two's complement is the universal representation for integer types[P0907R4], reflecting the reality of today's machine architectures. Unfortunately, changing the semantics of the built-in integer *operations* to reflect this is problematic for reasons of backward compatibility. This burden of C++'s long history can not be changed. What is arguably more problematic though is that the built-in operations are the *only* means of performing integer arithmetic in standard C++. This is particularly painful for library writers that need more precisely defined semantics for the integer operations, which match the operations provided by the underlying hardware.

This paper attempts to address the issue by proposing a number of library functions for performing basic integer arithmetic. The hope is that such a library will provide important building blocks to writers of libraries that rely heavily on integer arithmetic and serve as a useful stepping stone for standard library implementers to provide efficient implementations of future standard library facilities relying on integers coming out of SG6.

2 Motivation

Writing code that heavily relies on complex integer arithmetic is very hard in C++. The fact that all integer operations are subject to undefined behavior on overflow makes writing such code akin to dancing on the edge of a volcano. This leads to the bizarre situation where writing overflow aware code in C++ is actually harder than it is to write the equivalent code directly in assembly. The resulting C++ code is difficult to understand for both humans and compilers, and often leads to sub-optimal machine code being generated by the compiler.

Table 1 demonstrates this through the example of implementing an addition function that detects and reports integer overflow to the caller without causing undefined behavior. Implementing such a function correctly in C++ today is a significant challenge and most developers will probably get it wrong on their first try[1].

The generated machine code is as convoluted as the C++ code would suggest, despite the fact that the hardware performs the overflow check for free on every addition – integer overflow is not undefined behavior in assembly, but an expected result of the addition. We can simply carry out the `add` operation in assembly and check afterwards whether an overflow did occur. With the functionality proposed in this paper, developers will have access to a function that is similarly easy to use. The resulting code is much shorter, clearer and easier to reason about and also more closely resembles the nature of the operation that will be carried out by the underlying hardware.

[1] The author does not claim that the attempt provided here is correct

Table 1: Before-after table for an integer addition with overflow check.

Before

```cpp
#include <limits>

struct Result {
  int64_t sum;
  bool overflow;
};

Result safe_add(int64_t a, int64_t b) {
  using lim = std::numeric_limits<int64_t>;
  auto const max = lim::max();
  auto const min = lim::min();
  bool overflow = false;
  if (a >= 0) {
    if (max - a < b) {
      overflow = true;
    }
  } else {
    if (b < min - a) {
      overflow = true;
    }
  }
  return overflow ?
    Result{ .sum = 0, .overflow = true } :
    Result{ .sum = a+b, .overflow = false };
}
```

After

```cpp
#include <integer>

struct Result {
  int sum;
  bool overflow;
};

Result safe_add(int64_t a, int64_t b) {
  auto const [sum, overflow] =
    std::integer_i64_add_overflow(a, b);
  return Result{ .sum = sum,
                 .overflow = overflow };
}
```

x86-64 ASM Before

```
safe_add(long, long):
        test    rdi, rdi
        js      .L2
        movabs  rax, 9223372036854775807
        sub     rax, rdi
        cmp     rax, rsi
        jl      .L6
.L4:
        xor     ecx, ecx
        lea     rax, [rdi+rsi]
        movzx   edx, cl
        ret
.L2:
        movabs  rax, -9223372036854775808
        sub     rax, rdi
        cmp     rax, rsi
        jle     .L4
.L6:
        mov     ecx, 1
        xor     eax, eax
        movzx   edx, cl
        ret
```

x86-64 ASM After

```
safe_add(long, long):
        mov     rax, rdi
        mov     edx, 0
        add     rax, rsi
        seto    dl
        ret
```

3 Related Work

Functionality like the one being proposed in this paper has been discussed in SG6 before, in particular in the context of [P0103R1] *Overflow-Detecting and Double-Wide Arithmetic Operations*. That paper was last discussed in the context of the omnibus paper [P1889R1] *C++ Numerics Work In Progress*, which was criticized by SG6 for omitting all motivation and rationale of the earlier papers. The goal of this paper is to address these criticisms and restart the discussion for the overflow-detecting and double-width operations. The interface proposed in this paper is different from P0103 in that this proposal tries to mimic the underlying hardware as closely as possible (see 5.1) and proposes a different library interface. The author highly encourages critical feedback of this proposal in the context of the earlier work in P0103.

The C committee WG14 decided to ship facilities for safe integer arithmetic in C23, as laid out in [N2683] and [N2792]. These facilities provide a subset of what is proposed in this paper, focusing exclusively on the use case of safe integer arithmetic, which is concerned primarily with overflow detection. The author believes that a successful standard library solution should take additional use cases into account, besides safety. In particular, double-width operations should be considered as part of the design from the beginning.

Many modern programming languages already provide facilities like the ones being proposed in this paper. Both Swift[Swift-i32] and Rust[Rust-i32] provide low-level integer operations as part of their standard libraries and are able to generate superior machine code compared to standard C++ for uses like overflow detection[2].

Some compilers already provide parts of the functionality being proposed here through implementation-specific intrinsics[icc-Intr] or builtin functions[gcc-OF]. For the analysis of operations in 5, we only looked at functions supported by gcc. Compilers today are typically[3] not able to detect patterns for implementing overflow detection or double-width operations in standard C++ and generate sub-optimal machine code in those cases. In contrast, code generation for C++ code using the aforementioned intrinsics or builtins seems to be quite good in compilers that support them.

4 Use Cases

This section collects a number of essential use cases that a low-level integer arithmetic library is required to support.

4.1 Safe Integer Library

The goal of a safe integer library is to prevent errors resulting from invalid integer operations – in particular, preventing undefined behavior from signed integer overflow.

A safe integer library will want to detect for each operation whether it exceeded the valid range and force the user to handle the range error before continuing the calculation. Apart from the obvious overflow conditions when adding or multiplying big numbers, such a library would also want to detect fringe conditions like multiplying `INT_MIN` with -1.

[2] https://godbolt.org/z/8xjhfndd4 demonstrates overflow detection via the overflow flag on x86 in Rust

[3] The notable exception here being the use of double-width division machine instructions for code calculating both quotient and remainder of two integers. This seems to be a common optimization in modern compilers.

4.2 Big Integer Library

Big Integers provide data types and operations for performing computations that exceed the value range of the built-in integer types. This could include fixed-width integer types such as 512 bit integers, as well as dynamically sized *BigNum* types that can hold arbitrary large integer values. The algorithms for the basic arithmetic operations in a positional numeral system generalize to any base. A naïve implementation can encode an n bit number as an array of n `bool` values and implement the algorithms in a bit-by-bit fashion (base 2). The resulting implementation will be correct, but not very efficient.

For a more efficient implementation, the full word-width of the underlying machine should be used for each operation. On a 64 bit target machine, the algorithms will effectively be carried out in a positional system of base 64. For example, adding two n bit numbers is done by performing a sequence of 64 bit integer additions, carrying the overflow as a carry bit from lower integers to higher integers.

Addition of two n bit numbers will produce a result of $n + 1$ bits. Regardless of the base of the positional system that the addition is carried out in, an overflow will at most produce a single carry bit on overflow.

Efficient implementation of big integer addition requires an elemental integer addition operation that behaves as follows:

- Addition should wrap around on overflow. There should be no undefined behavior on overflow.
- It should be detectable whether an addition produced a *carry-bit* for unsigned and signed addition.
- It should be possible to feed the carry as an additional input to subsequent additions of the higher-order bits (*add-with-carry*)

Multiplication of two n bit numbers will produce a result of $2n$ bits. Carrying out long multiplication in a positional system requires adding up a number of double-width results, one for each position in the input number.

Efficient implementation of big integer multiplication requires a big integer addition operation as described above, as well as an elemental double-width integer multiplication that behaves as follows:

- Multiplication will produce a result that is twice the width of the width of its inputs. This result will be stored in two result variables that form the lower half and the higher half of the result value. For example, multiplying two 64 bit integers will produce a 128 bit result value, which will be split into two 64 bit return values.
- All results of any multiplication can be represented in the double-width result. There are no overflow conditions and there is no undefined behavior.

5 Operations

This section will define the desired properties for the arithmetic operations to fulfil the requirements for the use cases laid out in 4.

For each proposed operation we list prior art from functions in the Rust[Rust-i32] and Swift[Swift-i32] standard libraries, functions proposed in [P0103R1], and builtins[gcc-OF]/intrinsics[icc-Intr]

available in gcc.

We also discuss the machine instructions that would be emitted for the function on x86-64[x86-64], ARM64[ARM64], and RISC-V[RISCV] respectively. The author believes that those three architectures form a sufficiently representative sample of modern day hardware architectures, but welcomes any feedback about architectures that differ significantly from what is described.

5.1 General Assumptions and Design Guidelines

Operations should be well-defined for all inputs, where possible, and not exhibit any undefined behavior.

Operations should mimic the behavior of underlying machine instructions as closely as possible, similar to what one would expect from an intrinsic or builtin function.

When possible, operations should be carried out without any loss of information on overflow. In cases where deviation from this guideline could result in a performance benefit, an additional variant of the operation that loses information may be provided.

Operations are not supposed to be commonly used in user code. These are low-level facilities for library writers. As such, short names are not a priority. Names should be descriptive and unambiguous first; readability comes second.

5.2 Addition

5.2.1 Overflow Add (`add_overflow`)

Addition of two n bit numbers will result in an $n + 1$ bit number. The overflowing add is an operation that returns the wrapped around sum and an overflow bit indicating whether a wrap around did occur during addition.

Rust	Swift	P0103	gcc
`overflowing_add`	`addingReportingOverflow`	`overflow_add`	`__builtin_add_overflow`

All hardware architectures allow cheap checking of the overflow condition, either by inspection of a flag or via branch instructions. Overflow is not considered exceptional and does not trigger a trap. In hardware, the same instruction is used for signed and unsigned addition, which is why the check for the overflow condition is different[4]. For example, on x86 unsigned overflow is reported in the carry flag (CF), while signed overflow is reported in the overflow flag (OF). Since the unsigned overflow condition has no real meaning for signed types (and vice versa) we only report the overflow condition matching the input types.

x86-64	ARM64	RISC-V
Flags set on `add` (OF, CF)	Flags set on `add` (C, V)	Branch after add (`bltu`, `blt`)

Rust provides a number of additional operations for integer addition, which can be efficiently implemented based on the overflowing add:

- `checked_add` will return the sum wrapped in an `optional` which will be empty if the addition overflowed. This is trivial to implement on top of overflowing addition.

[4]For example, consider the signed operation of adding -3 to 5, which will result in 2. In unsigned arithmetic, this would be an overflow, as the binary representation of -3 in two's complement corresponds to a very big unsigned number. In signed arithmetic, the operation is perfectly fine and nowhere near an overflow.

- `saturating_add` will saturate the addition at `INT_MAX`/`INT_MIN` boundaries. This operation is not typically supported in hardware and will require additional branches in machine code, so a naïve implementation based on overflowing add should be sufficient for this.
- `wrapping_add` will just contain the wrapped result without the overflow indicator. Compilers are able to generate optimal machine code for this operation from a naïve implementation based on overflowing add[5], as it is easy to detect when the overflow condition is ignored.

Such functions could still be added later for reasons of usability but will not be part of this proposal.

5.2.2 Add-with-carry (`add_with_carry`)

When adding big numbers, an overflow from the lower bits has to be added as carry to the higher bits. Some hardware architectures offer a special *add-with-carry* operation for this purpose, which automatically take the carry flag set by earlier operations as input. Compilers supporting this operation through intrinsics are already able to generate the optimal machine code for such cases[6].

The result of add-with-carry is the same as for overflowing add. Adding two n bit numbers and a carry bit will result in an $n + 1$ bit number.

Rust	Swift	P0103	gcc
`carrying_add` (experimental)	—	`wide_add2`	`_addcarry_u32` (<x86intrin.h>)

Not all hardware architectures support add-with-carry natively. In particular, architectures that do not rely on flag registers for indicating overflow, such as RISC-V, may not benefit from distinguishing this operation.

x86-64	ARM64	RISC-V
`addc`	`adc`	—

5.3 Subtraction

5.3.1 Overflow Subtraction (`sub_overflow`)

For most intents and purposes, subtraction can simply be interpreted as adding a negative number. In a two's complement based architecture the concerns for subtraction are identical with those for addition.

Rust	Swift	P0103	gcc
`overflowing_sub`	`subtractingReportingOverflow`	`overflow_sub`	`__builtin_sub_overflow`

x86-64	ARM64	RISC-V
Flags set on `sub` (OF, CF)	Flags set on `sub` (C, V)	Branch after sub (`bltu`, `blt`)

5.3.2 Subtract-with-borrow `sub_with_borrow`

Subtract-with-borrow is the equivalent to additions add-with-carry.

Rust	Swift	P0103	gcc
`borrowing_sub` (experimental)	—	`wide_sub2`	`_subborrow_u32` (<x86intrin.h>)

[5] https://godbolt.org/z/n7cfcn6ss, note how this emits an `lea` instruction which does not set any of the overflow flags (otherwise the slower `add` will be generated)

[6] https://godbolt.org/z/nEKx78P5b, note the use of `add` followed by `addc`.

x86-64	ARM64	RISC-V
`sbb`	`sbc`	—

5.4 Multiplication

5.4.1 Overflow Multiplication (`mul_overflow`)

Multiplying two n bit numbers will result in a $2n$ bit number.

Modern hardware can typically carry out a full (double-width) multiplication in the same time as a single-word multiplication. However, the fact that the full multiplication will need two registers to write out its results may have detrimental effects on performance, which is why there is typically separate instructions for single and double-width multiplications.

In principle a compiler should be able to detect whether the high bits of a double-width operation remain unused and emit the single-width operation in machine code for such cases. However, a compiler would still need to be able to distinguish the case where the higher order bits are only accessed to check for overflow (which can be done most efficiently by a single-width multiply) from the case where the full value of the higher order bits is used.

This paper proposes a separate function for overflowing multiply (in addition to double-width multiply), which is consistent with how all the surveyed prior art approached the problem.

Rust	Swift	P0103	gcc
`overflowing_mul`	`multipliedReportingOverflow`	`overflow_mul`	`__builtin_mul_overflow`

RISC-V does not include multiplication in its Base Integer Instruction Set, as it can be emulated using additions and shifts. It does provide the standard extension *"M" for Integer Multiplication and Division*, which we surveyed here.

Unlike addition, multiplication can benefit from separate hardware instructions for signed and unsigned.

x86-64	ARM64	RISC-V
`mul`/`imul`	`umull`/`smull`	`mul`

5.4.2 Double-Width Multiply (`mul_wide`)

Carrying out multiplication without loss of information produces a result that is twice the length of the input. This means the result is returned as a pair of values (low and high).

Rust	Swift	P0103	gcc
`widening_mul` (experimental, unsigned only)	`multipliedFullWidth`	`wide_mul`	—

In hardware, the results are typically split across two registers. On some architectures, instead of a single multiply operation writing to two output registers, the multiplication operation is split into two instructions, one for writing out the lower half of the result and one for writing out the upper half. On such architectures, matching low and high instructions are usually fused into a single multiply operation internally.

x86-64	ARM64	RISC-V
`mul`/`imul`	`umulh`/`smulh`	`mulh`

5.4.3 Multiply with Carry (`mul_with_carry`)

For big integers, it may be beneficial to provide a double width multiply with carry.

Rust	Swift	P0103	gcc
`carrying_mul` (experimental, unsigned only)	—	`wide_muladd`	—

The author expects this to cause a higher burden on the implementation and would appreciate feedback as to how important such operations are deemed by SG6.

5.5 Division

5.5.1 Overflow Division (`div_overflow`)

Division by 0 is undefined and will not produce a valid result.

Division can overflow only for the degenerate case `INT_MIN`$/-1$. This is often not indicated correctly by the hardware and may require branching in machine code for efficient implementations.

Both cases should be diagnosed correctly by an overflowing division operation.

Rust	Swift	P0103	gcc
`overflowing_div`	`dividedReportingOverflow`	`overflow_div`	—

5.5.2 Overflow Remainder (`rem_overflow`)

The error conditions for remainder are identical to those for division.

Rust	Swift	P0103	gcc
`overflowing_rem`	`remainderReportingOverflow`	—	—

5.5.3 Fused Division-Remainder (`div_rem_overflow`)

Carrying out division in hardware usually computes the remainder for free. As with multiplication, returning the remainder in a second register may have a negative impact on performance.

A fused division-remainder is subject to the same error condition as the overflow division, so unlike other wide operations, it is not guaranteed to produce a valid result.

Rust	Swift	P0103	gcc
—	—	`wide_divnrem`	—

Many compilers are already able to fuse matching division and remainder operations to a single machine instruction.

5.5.4 Double-Width Division (`div_wide`)

Some hardware architectures allow the dividend to be double-width. A double-width division is still subject to overflow and division-by-0 errors.

Rust	Swift	P0103	gcc
—	—	`wide_divn`	—

Hardware support for double-width division is not common.

x86-64	ARM64	RISC-V
`div`	—	—

5.6 Other operations

5.6.1 Overflowing Negation (`neg_overflow`)

Negation can overflow for `INT_MIN`.

Rust	Swift	P0103	gcc
`overflowing_neg`	—	`overflow_neg`	—

5.7 Prioritization Assessment

The author acknowledges that it may not be reasonable to add the full set of operations discussed in this section all at once to the standard library. Not all operations are equally useful, and not all operations are equally easy to implement for compiler vendors.

In this section, we propose a prioritization for the individual operations to distinguish between the ones that we consider essential for this proposal to bring any value at all, and those that would just be nice-to-have. These sets may be proposed separately in different papers in the future to accelerate adoption.

The first set is the operations that we consider essential. All of these are universally useful, are supported by all modern hardware architectures and there is broad implementation experience in other languages for these. We are confident that they can be shipped quickly with minimal effort:

- `add_overflow`
- `sub_overflow`
- `mul_overflow`

The second set contains operations that are equally easy to implement, but may not be quite as useful or important to library writers. In particular, these functions are much easier to implement manually without standard library support:

- `div_overflow`
- `rem_overflow`
- `neg_overflow`

The next set contains operations that would be useful for some use cases, but may be more difficult to implement and require additional implementation experience and/or feedback from implementers before standardization:

- `mul_wide`
- `div_rem_overflow`
- `sub_with_borrow`
- `div_wide`
- `add_with_carry`
- `mul_with_carry`

6 Library Interface

6.1 Return Values or Out Parameters

Both Swift and Rust have language-level support tuples and rely on this mechanism for operations that produce multiple return values (overflow flag + value; low return + high return).

P0103 and the gcc builtins use output parameters and single return values. For instance, in case of an overflow function, the `bool` return value indicates whether an overflow occurred, and the wrapped sum is written to an output parameter of the function.

The intention is that the proposed functions will be used similar to the gcc builtin functions, so for the relevant use cases of this proposal the decision between return values and out parameters should not have an impact on performance.

The author slightly favors a design using aggregate types as return values.

6.2 Overloads, Function Templates, or Named Functions

There is a question how operations on different types should be grouped together. The previous paper P0103 proposed providing the functionality through function templates. This author feels that such an interface is potentially confusing in that it is difficult for users to tell which specializations will be provided by the standard library. A negative example here is `std::atomic<T>` which commonly confuses users as to which types can sensibly be used with an atomic.

An alternative would be to provide a common set of overloads for each function. This author dislikes the solution because it requires users of this library to understand both overload resolution and the implicit integer conversion rules to ensure that the correct operation will be selected. We believe that the typical user of this library will not necessarily be an expert of the language and may find it troublesome to deal with those issues.

We therefore propose a each function under its own name, where the type of the target operands is part of the name, similar to the names of the gcc builtin functions. This approach leads to code that is very explicit, with no confusion about what the target operands for the operation are. It also makes it easy to extend the set of operations later, as each new function will use a new unique name that does not interfere with existing functions. The major downside of this approach is that it makes the functions tedious to use directly in generic contexts. As we consider them to be low-level building blocks for library authors, we are confident that authors of generic code will have no trouble wrapping those functions in an interface that is more suited for such uses, such as the two options described previously.

6.3 Nomenclature

The author proposes the following nomenclature:

- All functions share the common prefix `integer_`
- Following the prefix is one of the type identifiers: `i8_`, `u8_`, `i16_`, `u16_`, `i32_`, `u32_`, `i64_`, or `u64_`
- Followed by the name of the operation.

We propose the following names for the individual operations:

- `add_overflow`
- `sub_overflow`
- `mul_overflow`
- `div_overflow`
- `rem_overflow`
- `neg_overflow`
- `mul_wide`
- `add_with_carry`
- `sub_with_borrow`
- `mul_with_carry`
- `div_rem_overflow`
- `div_wide`

The full name for the function performing overflow addition of two `int32_t` signed 32 bit signed integer values would be `std::integer_i32_add_overflow`.

The return type of each function is the name of the function with an appended `_result`.

6.4 Header

The proposal suggests to create a new header `<integer>` for all of the proposed functions. The suggested feature test macro is `__cpp_lib_low_level_integer`.

6.5 Example Signature

```
namespace std {
  struct integer_i32_add_overflow_result {
    int32_t sum;
    bool does_overflow;
  };
  integer_i32_add_overflow_result integer_i32_add_overflow(int32_t a, int32_t b);
}
```

7 Directional Feedback

7.1 Questions to SG6

The author would like to ask SG6 for directional feedback on the following questions:

- Do we agree that the standard library should provide some form of low-level integer arithmetic, as laid out in the motivational sections of this paper?
- Do we see any additional use cases that we consider essential, besides the ones described in 4?
- Do we agree with the general assumptions and design guidelines proposed for such a library in 5.1?
- Do we see any additional operations that could be useful to standardize in the context of low-level integer arithmetic that were not mentioned in 5?
- Do we agree with the prioritization assessment given in 5.7?

7.2 Questions to LEWG

The author would like to ask LEWG/SG18(LEWG-I) for directional feedback on the following questions:

- Do we agree that the standard library should provide some form of low-level integer arithmetic, as laid out in the motivational sections of this paper?
- Do we agree with the interface direction proposed in 6?
  - Do we prefer aggregate return types over output parameters?
  - Do we prefer named functions to overloaded functions or function templates?
  - Do we agree with the proposed naming scheme?

Acknowledgements

The author would like to thank Robert C. Seacord for the feedback and discussions on the topic of this paper.

References

[ARM64] *Arm Architecture Reference Manual for A-profile architecture*
https://developer.arm.com/documentation/ddi0487/latest/

[x86-64] *Intel® 64 and IA-32 Architectures Software Developer's Manual*
https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[RISCV] *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*
https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view?usp=drive_link

[gcc-OF] *Built-in Functions to Perform Arithmetic with Overflow Checking*
https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html

[icc-Intr] *Intel C++ Compiler Intrinsics for Multi-Precision Arithmetic*
https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/intrinsics-for-multi-precision-arithmetic.html

[Rust-i32] *Rust Primitive Type `i32`*
https://doc.rust-lang.org/std/primitive.i32.html

[Swift-i32] *Swift `Int32` Type*
https://developer.apple.com/documentation/swift/int32

[P0103R1] *Lawrence Crowl – Overflow-Detecting and Double-Wide Arithmetic Operations*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0103r1.html

[P0907R4] *JF Bastien – Signed Integers are Two's Complement*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0907r4.html

[P1889R1] *Alexander Zaitsev, Antony Polukhin – C++ Numerics Work In Progress*
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1889r1.pdf

[N2683] *David Svoboda – Towards Integer Safety*
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2683.pdf

[N2792] *David Svoboda – Supplemental Integer Safety*
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2792.pdf