Document: P1998R0
Author: Ryan McDougall <mcdougall.ryan@gmail.com>
Audience: LEWG-I, SG-6
Project: ISO/IEC JTC1/SC22/WG21 14882: Programming Language — C++

Simple Facility for Lossless Integer Conversion

Abstract

Programmers usually think in mathematical terms, such as natural numbers or integers, rather than signed or unsigned machine words with under or over-flow, and the majority of values found in integer variables in most programs will not test those distinctions.

If programmer assumptions about integer operations in their program no longer hold, and there is a mismatch expected and implementation behavior, it is a sign of a software defect, and it is ideal that a fault be raised as early as possible.

Motivation

Critical systems should be reliable. Even non-critical systems should avoid defects. Data loss or undefined behavior caused by integer truncation, underflow, or overflow, are a common sources of software defects. For this reason many critical coding standards, as part of a safety or reliability regimen, place clear restrictions valid operations with integer data. For Autosar 14 this means:

1. Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types **[A3-9-1]**
2. An integer expression shall not lead to data loss **[A4-7-1]**
3. [An] expression shall not be implicitly converted to a different underlying type **[M5-0-3]**
4. *An explicit integral conversion shall not change the signedness of the underlying type of [an] expression **[M5-0-9]***
5. Evaluation of constant unsigned integer expressions shall not lead to wrap-around **[M5-19-1]**

Mixed Operations

It is not possible or desirable prohibit sign conversion in general. Different libraries have different opinions on how to represent counting data. An unsigned integer might superficially model a natural number, wrap around on under/over-flow does not. Perhaps negative numbers could usefully represent special sentinel values? Conversions are common and must be handled.

```
bool foo(int index, std::vector<bool> v) {
  if (index >= 0 && index < v.size()) { // -- mixed comparison
    return v[i];                        // -- conversion to
unsigned
  } else {
    // special case
  }
}
```

Explicit Narrowing

Most of our counting values are small and positive, and are represented as the same bit pattern on signed and unsigned words. Explicit narrowing conversions for these values is usually safe. However if data is lost in conversion, it is a sign of a defect elsewhere in the program.

```
int foo();
uint8_t bear_count = foo();  // -- never more than 255 bears
for (size_t i=0; i < bears_count; ++i) {
  bears[i].eat(honey);
}
```

In this case implicit contract violation is observed as very unfortunate bear starvation.

Over/Under-flow

Any integer operation could overflow or underflow (with well or undefined behavior).

```
int8_t bar(int8_t x, int8_t y) {
  return x + y;  // -- possible overflow
}
int8_t baz(uint8_t x, uint8_t y) {
  return x - y;  // -- possible underflow
}
int8_t zif(uint8_t x, uint8_t y) {
  return x * y;  // -- possible wrap around
}
```

In practice, if arithmetic operands have tight bounds, it's because the values are not expected to exceed the word size. If an operation over/under-flows it's a sign of a defect elsewhere in the program.

Coding Standard Compliance

If the programmer defensively checks the values and determines data loss cannot occur under the operation, then the code is compliant.

```
int v = foo();
assert(v >= 0 && v <= 255);
uint8_t x = v;  // -- [A4-7-1] compliant

assert(x <= 123);
uint8_t y = bar(x, 4);  // -- [A4-7-1] compliant
```

Suggested Facility

The following participate in overload resolution only if their types are integral.

Detecting Information Loss

The names denote that we test the operand *values* for validity in their destination type, after promotion and conversion.

```
template <typename To, typename U>
constexpr bool std::is_value_lossless_convertable(U from);

// For exposition-only signed integer type V,
//  capable of holding all values of To and U
// As-if:
//  const auto v = static_cast<V>(from);
//  return v >= std::numeric_limits<To>::min() &&
//         v <= std::numeric_limits<To>::max();

template <typename To, typename T, typename U>
constexpr bool std::is_value_lossless_addable(T t, U u);

// For exposition-only signed integer type V,
//  capable of holding all values of To,
//  and all the values of T and U added together
// As-if:
//  const auto v = static_cast<V>(t) + static_cast<V>(u);
//  return v >= std::numeric_limits<To>::min() &&
//         v <= std::numeric_limits<To>::max();

template <typename To, typename T, typename U>
constexpr bool std::is_value_lossless_subtractable(T t, U u);
```

```
// For exposition-only signed integer type V,
//  capable of holding all values of To,
//  and all the values of T subtracted with U
// As-if:
//  const auto v = static_cast<V>(t) - static_cast<V>(u);
//  return v >= std::numeric_limits<To>::min() &&
//         v <= std::numeric_limits<To>::max();

template <typename To, typename T, typename U>
constexpr std::is_value_lossless_multipliable(T t, U u);

// For exposition-only signed integer type V,
//  capable of holding all values of To,
//  and all the values of T and U multiplied together
// As-if:
//  const auto v = static_cast<V>(t) * static_cast<V>(u);
//  return v >= std::numeric_limits<To>::min() &&
//         v <= std::numeric_limits<To>::max();
```

Division is not considered because integer division is less common, and inherently lossy.

The detection functions are the core of this proposal, since implementing the logic required to promote and convert all operands at each step correctly would be difficult for most users, and result in hard to discover errors. Yet for implementations it would be straight forward to ensure a fast and correct version.

Checked Conversions

Purpose driven narrowing conversions are offered explicitly in service of the most common case, and directly addresses rules **[A4-7-1]** and **[M5-0-3]**.

```
template <typename From, typename To>
To std::narrow(From from) noexcept;

// As-if:
// if (is_value_lossless_convertable<To>(from)) {
//   return static_cast<To>(from);
// } else {
//   implementation defined terminate
// }

template <typename From, typename To>
To std::narrow_cast(From from) noexcept(false);

// As-if:
```

```
// if (is_value_lossless_convertable<To>(from)) {
//   return static_cast<To>(from);
// } else {
//   throw narrow_error{implementation-defined};
// }

template <typename From, typename To, typename Handler>
To std::narrow_or(From, Handler&& alternative) noexcept;

// As-if:
// if (is_value_lossless_convertable<To>(from)) {
//   return static_cast<To>(from);
// } else {
//   return handler(from);
// }
```

Checked Arithmetic

Programmers prefer to use arithmetic operators directly, and code that doesn't quickly becomes difficult to read. Without a type-safe integer type to overload an operator on, this proposal offers no convenience library for checked addition, subtraction, or multiplication.

Checked Literals

Checking early means checking at compile time as well. This addresses rules **[A3-9-1]**, **[A4-7-1]** and **[M5-0-3]**.

```
constexpr int8_t  operator"" _i8(implementation-defined);
constexpr int16_t operator"" _i16(implementation-defined);
constexpr int32_t operator"" _i32(implementation-defined);
constexpr int64_t operator"" _i64(implementation-defined);
...
constexpr uint8_t  operator"" _u8(implementation-defined);
constexpr uint16_t operator"" _u16(implementation-defined);
constexpr uint32_t operator"" _u32(implementation-defined);
constexpr uint64_t operator"" _u64(implementation-defined);
...

// For checked literal T
// As-if:
// static_assert(is_value_lossless_convertable<T>(from-impl-defn));
```

Comparison Table

Before	After
`// unsafe` `uint16_t v = foo();`	`auto v =` `  std::narrow_cast<uint16_t>(foo());`

`// manual`
`uint16_t v;`
`auto t = foo();`
`if (t >= 0 && t < 200) {`
`  v = static_cast<int16_t>(t);`
`} else {`
`  assert(false);`
`}`
	`auto v = std::narrow<uint16_t>(foo());`

`// defaulted`
`uint16_t v = -1;`
`auto t = foo();`
`if (t >= 0 && t < 200) {`
`  v = static_cast<int16_t>(t);`
`}`
	`auto v =`
	`std::narrow_or<uint16_t>(foo(),`
	`    [](...){ return -1; });`

`// incorrect ...`
`int16_t v;`
`uint64_t t = foo();`
`if (t >= -200 && t < 200) {`
`  v = static_cast<int16_t>(t);`
`}`
	`uint64_t t = foo();`
	`auto v = std::narrow_cast<uint16_t>(t);`

`// contract`
`auto t = foo();`
`assert(`
` t <=`
` std::numeric_limits<uint16_t>::min()`
`&&`
` t >=`
` std::numeric_limits<uint16_t>::max()`
`);`
	`auto t = foo();`
	`assert(is_value_lossless_convertable`
	`    <uint16_t>(t));`

`// implicit conversion literal` `constexpr uint16_t v = -5;`	`// compile error literal` `constexpr uin16_t v = -5_u16;`

Existing Work

User Libraries

It is possible to implement this facility without direct support from the standard or implementations. However for many users it would be difficult and error prone, with hard to spot bugs or poor performance.

There are larger libraries for integer type-safety available, but the facility proposed here could be considered basis functionality, and would suit the needs of most users.

Contracts

Motivation has been presented in terms of expected behavior relative to implicit contracts. When C++ has first class explicit contracts, the data loss detection facilities can be used to in contract pre- or post- condition specification, and the narrowing facilities can be re-implemented in terms of contract pre-conditions.

Functions for Testing Boundary Conditions on Integer Operations [P1619]

My reading of the following:

“The result of the expression and the result of the mathematical operation would be congruent modulo2N. Further, for functions not ending in “_modular”, the result of the expression and the result of the mathematical operation would be equal.”

suggests that this paper’s `is_value_lossless_convertable<To>` is equivalent to P1619’s `can_convert<To>`, and this paper’s `is_value_lossless_addable<To> is equivalent to `can_convert<To>(can_add)`, etc.. If P1619 is assumed, this paper can devolve into offering checked conversions and checked literals.

Numeric Traits for the Standard Library [P0437]

Proposes constant values, and does not check values at runtime.

Composition of Arithmetic Types [P0554]

Proposes a broad type-safe facility based on composition of vocabulary types. However it presumes all user code is implemented using the library. There will remain a large amount of user code in terms of fundamental integer types.

C++ Numerics Work In Progress [P1889]

Omnibus work that appears to incorporate a number of the above proposals into one for the purpose of publishing as a TS.