base/numerics

This directory contains a dependency-free, header-only library of templates providing well-defined semantics for safely and performantly handling a variety of numeric operations, including most common arithmetic operations and conversions.

The public API is broken out into the following header files:

- `checked_math.h` contains the `CheckedNumeric` template class and helper functions for performing arithmetic and conversion operations that detect errors and boundary conditions (e.g. overflow, truncation, etc.).
- `clamped_math.h` contains the `ClampedNumeric` template class and helper functions for performing fast, clamped (i.e. non-sticky saturating) arithmetic operations and conversions.
- `safe_conversions.h` contains the `StrictNumeric` template class and a collection of custom casting templates and helper functions for safely converting between a range of numeric types.
- `safe_math.h` includes all of the previously mentioned headers.

*** aside **Note:** The `Numeric` template types implicitly convert from C numeric types and `Numeric` templates that are convertable to an underlying C numeric type. The conversion priority for `Numeric` type coercions is:

- `StrictNumeric` coerces to `ClampedNumeric` and `CheckedNumeric`
- `ClampedNumeric` coerces to `CheckedNumeric`

[TOC]

Common patterns and use-cases

The following covers the preferred style for the most common uses of this library. Please don't cargo-cult from anywhere else. 😉

Performing checked arithmetic type conversions

The `checked_cast` template converts between arbitrary arithmetic types, and is used for cases where a conversion failure should result in program termination:

```
// Crash if signed_value is out of range for buff_size.
size_t buff_size = checked_cast<size_t>(signed_value);
```

Performing saturated (clamped) arithmetic type conversions

The `saturated_cast` template converts between arbitrary arithmetic types, and is used in cases where an out-of-bounds source value should be saturated to the corresponding maximum or minimum of the destination type:

```
// Convert from float with saturation to INT_MAX, INT_MIN, or 0 for NaN.
int int_value = saturated_cast<int>(floating_point_value);
```

Enforcing arithmetic type conversions at compile-time

The `strict_cast` emits code that is identical to `static_cast`. However, provides static checks that will cause a compilation failure if the destination type cannot represent the full range of the source type:

```
// Throw a compiler error if byte_value is changed to an out-of-range-type.
int int_value = strict_cast<int>(byte_value);
```

You can also enforce these compile-time restrictions on function parameters by using the `StrictNumeric` template:

```
// Throw a compiler error if the size argument cannot be represented by a
// size_t (e.g. passing an int will fail to compile).
bool AllocateBuffer(void** buffer, StrictCast<size_t> size);
```

Comparing values between arbitrary arithmetic types

Both the `StrictNumeric` and `ClampedNumeric` types provide well defined comparisons between arbitrary arithmetic types. This allows you to perform comparisons that are not legal or would trigger compiler warnings or errors under the normal arithmetic promotion rules:

```
bool foo(unsigned value, int upper_bound) {
  // Converting to StrictNumeric allows this comparison to work correctly.
  if (MakeStrictNum(value) >= upper_bound)
    return false;
```

*** note **Warning:** Do not perform manual conversions using the comparison operators. Instead, use the cast templates described in the previous sections, or the constexpr template functions `IsValueInRangeForNumericType` and `IsTypeInRangeForNumericType`, as these templates properly handle the full range of corner cases and employ various optimizations.

Calculating a buffer size (checked arithmetic)

When making exact calculations—such as for buffer lengths—it's often necessary to know when those calculations trigger an overflow, undefined behavior, or other boundary conditions. The `CheckedNumeric` template does this by storing a bit determining whether or not some arithmetic operation has occured that would put the variable in an "invalid" state. Attempting to extract the value from a variable in an invalid state will trigger a check/trap condition, that by default will result in process termination.

Here's an example of a buffer calculation using a `CheckedNumeric` type (note: the AssignIfValid method will trigger a compile error if the result is ignored).

```
// Calculate the buffer size and detect if an overflow occurs.
size_t size;
if (!CheckAdd(kHeaderSize, CheckMul(count, kItemSize)).AssignIfValid(&size)) {
  // Handle an overflow error...
}
```

Calculating clamped coordinates (non-sticky saturating arithmetic)

Certain classes of calculations—such as coordinate calculations—require well-defined semantics that always produce a valid result on boundary conditions. The `ClampedNumeric` template addresses this by providing performant, non-sticky saturating arithmetic operations.

Here's an example of using a `ClampedNumeric` to calculate an operation insetting a rectangle.

```
// Use clamped arithmetic since inset calculations might overflow.
void Rect::Inset(int left, int top, int right, int bottom) {
  origin_ += Vector2d(left, top);
  set_width(ClampSub(width(), ClampAdd(left, right)));
  set_height(ClampSub(height(), ClampAdd(top, bottom)));
}
```

*** note The `ClampedNumeric` type is not "sticky", which means the saturation is not retained across individual operations. As such, one arithmetic operation may result in a saturated value, while the next operation may then "desaturate" the value. Here's an example:

```
ClampedNumeric<int> value = INT_MAX;
++value;  // value is still INT_MAX, due to saturation.
--value;  // value is now (INT_MAX - 1), because saturation is not sticky.
```

Conversion functions and StrictNumeric<> in safe_conversions.h

This header includes a collection of helper `constexpr` templates for safely performing a range of conversions, assignments, and tests.

Safe casting templates

- `as_signed()` - Returns the supplied integral value as a signed type of the same width.
- `as_unsigned()` - Returns the supplied integral value as an unsigned type of the same width.
- `checked_cast<>()` - Analogous to `static_cast<>` for numeric types, except that by default it will trigger a crash on an out-of-bounds conversion (e.g. overflow, underflow, NaN to integral) or a compile error if the conversion error can be detected at compile time. The crash handler can be overridden to perform a behavior other than crashing.
- `saturated_cast<>()` - Analogous to `static_cast` for numeric types, except that it returns a saturated result when the specified numeric conversion would otherwise overflow or underflow. An NaN source returns 0 by default, but can be overridden to return a different result.
- `strict_cast<>()` - Analogous to `static_cast` for numeric types, except this causes a compile failure if the destination type is not large enough to contain any value in the source type. It performs no runtime checking and thus introduces no runtime overhead.

Other helper and conversion functions

- `IsValueInRangeForNumericType<>()` - A convenience function that returns true if the type supplied as the template parameter can represent the value passed as an argument to the function.
- `IsTypeInRangeForNumericType<>()` - A convenience function that evaluates entirely at compile-time and returns true if the destination type (first template parameter) can represent the full range of the source type (second template parameter).
- `IsValueNegative()` - A convenience function that will accept any arithmetic type as an argument and will return whether the value is less than zero. Unsigned types always return false.
- `SafeUnsignedAbs()` - Returns the absolute value of the supplied integer parameter as an unsigned result (thus avoiding an overflow if the value is the signed, two's complement minimum).

StrictNumeric<>

`StrictNumeric<>` is a wrapper type that performs assignments and copies via the `strict_cast` template, and can perform valid arithmetic comparisons across any range of arithmetic types. `StrictNumeric` is the return type for values extracted from a `CheckedNumeric` class instance. The raw numeric value is extracted via `static_cast` to the underlying type or any type with sufficient range to represent the underlying type.

- `MakeStrictNum()` - Creates a new `StrictNumeric` from the underlying type of the supplied arithmetic or StrictNumeric type.
- `SizeT` - Alias for `StrictNumeric<size_t>`.

CheckedNumeric<> in checked_math.h

`CheckedNumeric<>` implements all the logic and operators for detecting integer boundary conditions such as overflow, underflow, and invalid conversions. The `CheckedNumeric` type implicitly converts from floating point and integer data types, and contains overloads for basic arithmetic operations (i.e.: `+`, `-`, `*`, `/` for all types and `%`, `<<`, `>>`, `&`, `|`, `^` for integers). However, *the variadic template functions are the prefered API,* as they remove type ambiguities and help prevent a number of common errors. The variadic functions can also be more performant, as they eliminate redundant expressions that are unavoidable with the with the operator overloads. (Ideally the compiler should optimize those away, but better to avoid them in the first place.)

Type promotions are a slightly modified version of the standard C/C++ numeric promotions with the two differences being that *there is no default promotion to int* and *bitwise logical operations always return an unsigned of the wider type.*

Members

The unary negation, increment, and decrement operators are supported, along with the following unary arithmetic methods, which return a new `CheckedNumeric` as a result of the operation:

- `Abs()` - Absolute value.
- `UnsignedAbs()` - Absolute value as an equal-width unsigned underlying type (valid for only integral types).
- `Max()` - Returns whichever is greater of the current instance or argument. The underlying return type is whichever has the greatest magnitude.
- `Min()` - Returns whichever is lowest of the current instance or argument. The underlying return type is whichever has can represent the lowest number in the smallest width (e.g. int8_t over unsigned, int over int8_t, and float over int).

The following are for converting `CheckedNumeric` instances:

- `type` - The underlying numeric type.
- `AssignIfValid()` - Assigns the underlying value to the supplied destination pointer if the value is currently valid and within the range supported by the destination type. Returns true on success.
- `Cast<>()` - Instance method returning a `CheckedNumeric` derived from casting the current instance to a `CheckedNumeric` of the supplied destination type.

*** aside The following member functions return a `StrictNumeric`, which is valid for comparison and assignment operations, but will trigger a compile failure on attempts to assign to a type of insufficient range. The underlying value can be extracted by an explicit `static_cast` to the underlying type or any type with sufficient range to represent the underlying type.

- `IsValid()` - Returns true if the underlying numeric value is valid (i.e. has not wrapped or saturated and is not the result of an invalid conversion).

- `ValueOrDie()` - Returns the underlying value. If the state is not valid this call will trigger a crash by default (but may be overridden by supplying an alternate handler to the template).
- `ValueOrDefault()` - Returns the current value, or the supplied default if the state is not valid (but will not crash).

Comparison operators are explicitly not provided for `CheckedNumeric` types because they could result in a crash if the type is not in a valid state. Patterns like the following should be used instead:

```
// Either input or padding (or both) may be arbitrary sizes.
size_t buff_size;
if (!CheckAdd(input, padding, kHeaderLength).AssignIfValid(&buff_size) ||
     buff_size >= kMaxBuffer) {
  // Handle an error...
} else {
  // Do stuff on success...
}
```

Non-member helper functions

The following variadic convenience functions, which accept standard arithmetic or `CheckedNumeric` types, perform arithmetic operations, and return a `CheckedNumeric` result. The supported functions are:

- `CheckAdd()` - Addition.
- `CheckSub()` - Subtraction.
- `CheckMul()` - Multiplication.
- `CheckDiv()` - Division.
- `CheckMod()` - Modulus (integer only).
- `CheckLsh()` - Left integer shift (integer only).
- `CheckRsh()` - Right integer shift (integer only).
- `CheckAnd()` - Bitwise AND (integer only with unsigned result).
- `CheckOr()` - Bitwise OR (integer only with unsigned result).
- `CheckXor()` - Bitwise XOR (integer only with unsigned result).
- `CheckMax()` - Maximum of supplied arguments.
- `CheckMin()` - Minimum of supplied arguments.

The following wrapper functions can be used to avoid the template disambiguator syntax when converting a destination type.

- `IsValidForType<>()` in place of: `a.template IsValid<>()`
- `ValueOrDieForType<>()` in place of: `a.template ValueOrDie<>()`
- `ValueOrDefaultForType<>()` in place of: `a.template ValueOrDefault<>()`

The following general utility methods is are useful for converting from arithmetic types to `CheckedNumeric` types:

- `MakeCheckedNum()` - Creates a new `CheckedNumeric` from the underlying type of the supplied arithmetic or directly convertible type.

ClampedNumeric<> in clamped_math.h

`ClampedNumeric<>` implements all the logic and operators for clamped (non-sticky saturating) arithmetic operations and conversions. The `ClampedNumeric` type implicitly converts back and forth between floating point

and integer data types, saturating on assignment as appropriate. It contains overloads for basic arithmetic operations (i.e.: `+`, `-`, `*`, `/` for all types and `%`, `<<`, `>>`, `&`, `|`, `^` for integers) along with comparison operators for arithmetic types of any size. However, *the variadic template functions are the prefered API,* as they remove type ambiguities and help prevent a number of common errors. The variadic functions can also be more performant, as they eliminate redundant expressions that are unavoidable with the operator overloads. (Ideally the compiler should optimize those away, but better to avoid them in the first place.)

Type promotions are a slightly modified version of the standard C/C++ numeric promotions with the two differences being that *there is no default promotion to int* and *bitwise logical operations always return an unsigned of the wider type*.

*** aside Most arithmetic operations saturate normally, to the numeric limit in the direction of the sign. The potentially unusual cases are:

- **Division:** Division by zero returns the saturated limit in the direction of sign of the dividend (first argument). The one exception is 0/0, which returns zero (although logically is NaN).
- **Modulus:** Division by zero returns the dividend (first argument).
- **Left shift:** Non-zero values saturate in the direction of the signed limit (max/min), even for shifts larger than the bit width. 0 shifted any amount results in 0.
- **Right shift:** Negative values saturate to -1. Positive or 0 saturates to 0. (Effectively just an unbounded arithmetic-right-shift.)
- **Bitwise operations:** No saturation; bit pattern is identical to non-saturated bitwise operations.

Members

The unary negation, increment, and decrement operators are supported, along with the following unary arithmetic methods, which return a new `ClampedNumeric` as a result of the operation:

- `Abs()` - Absolute value.
- `UnsignedAbs()` - Absolute value as an equal-width unsigned underlying type (valid for only integral types).
- `Max()` - Returns whichever is greater of the current instance or argument. The underlying return type is whichever has the greatest magnitude.
- `Min()` - Returns whichever is lowest of the current instance or argument. The underlying return type is whichever has can represent the lowest number in the smallest width (e.g. int8_t over unsigned, int over int8_t, and float over int).

The following are for converting `ClampedNumeric` instances:

- `type` - The underlying numeric type.
- `RawValue()` - Returns the raw value as the underlying arithmetic type. This is useful when e.g. assigning to an auto type or passing as a deduced template parameter.
- `Cast<>()` - Instance method returning a `ClampedNumeric` derived from casting the current instance to a `ClampedNumeric` of the supplied destination type.

Non-member helper functions

The following variadic convenience functions, which accept standard arithmetic or `ClampedNumeric` types, perform arithmetic operations, and return a `ClampedNumeric` result. The supported functions are:

- `ClampAdd()` - Addition.
- `ClampSub()` - Subtraction.
- `ClampMul()` - Multiplication.

- `ClampDiv()` - Division.
- `ClampMod()` - Modulus (integer only).
- `ClampLsh()` - Left integer shift (integer only).
- `ClampRsh()` - Right integer shift (integer only).
- `ClampAnd()` - Bitwise AND (integer only with unsigned result).
- `ClampOr()` - Bitwise OR (integer only with unsigned result).
- `ClampXor()` - Bitwise XOR (integer only with unsigned result).
- `ClampMax()` - Maximum of supplied arguments.
- `ClampMin()` - Minimum of supplied arguments.

The following is a general utility method that is useful for converting to a `ClampedNumeric` type:

- `MakeClampedNum()` - Creates a new `ClampedNumeric` from the underlying type of the supplied arithmetic or directly convertible type.