Fixed-Point Math

This C++20 library introduces a fixed-point mathematical system designed specifically for embedded C++ systems. It simplifies integer math operations and incorporates compile-time and carefully optimized runtime checks to ensure that integers do not exceed predefined value ranges.

Motivation

In embedded systems programmed with C/C++, it is typical to avoid built-in floating-point types like `float` and `double` due to their significant impact on code size and execution time—critical factors in microcontrollers. A common solution is to use integers scaled by a fixed power of two, allowing representation of fractional values. For instance, a 32-bit unsigned integer scaled by the factor 2^{16} splits the integer into 16 bits for the whole numbers (0-65535) and 16 bits for the fractional part, offering a precision of 2^{-16}. This `UQ16.16` fixed-point type, prevalent in ARM's Q notation, can represent values from `0.0` to `65535.99998474`. Negative values are typically handled using signed integers in two's complement form, like the `Q14.2` type, which spans from `-8192.00` to `8191.75`.

Problems and Expectations

While the concept of fixed-point math is straightforward, it often becomes a source of bugs: - Mathematical operations, especially multiplications and complex formulas, can quickly exceed the value range of the base type, leading to silent overflows and unpredictable results if no overflow checks are in place. - Manual calculations with scaled values are cumbersome and error-prone as scaling corrections are easily overlooked due to their non-intuitive nature.

The goal is to enable calculations within a specified value range and precision without constant concerns about overflow. Here are the features expected from a well-designed fixed-point library which tackles the aforementioned problems:

- Predefined fixed-point types based on both signed and unsigned integer types (8, 16, 32, and possibly 64 bits).
- User-defined precision and value range set at compile-time.

- Specification of value range using float values at compile-time for intuitive setup.
- Capability to adjust precision and/or value range later in the code, still at compile-time.
- Prohibit implicit construction from floating-point types to prevent ambiguity.
- Support for implicit construction from static integers in formulas (current limitations prevent constexpr function arguments; literals are used instead).
- Explicit construction from both scaled integers and real floating-point values at compile-time.
- Explicit construction from integer-based variables with scaled values at runtime.
- Disallow runtime construction from floating-point variables to maintain integrity in resource-constrained environments.
- Implement compile-time overflow checks where feasible, with runtime checks as necessary.
- Various overflow handling strategies (e.g., error, assert, clamp, or allow unchecked overflow).
- Implicit conversions between types of the same base type only if it results in higher precision; explicit conversion for down-scaling to lower precision.
- Conversion between different base types only through explicit casts (static cast, safe cast, force cast).
- Straightforward, easy-to-debug formulas without obscuring scaling corrections.
- Implementation of standard mathematical operations and specialized functions such as square, sqrt, cube and cbrt using adapted formulas to maintain integer-only calculations.
- Advanced mathematical operators and functions, like reciprocal square roots, implemented to accommodate the fixed-point calculation model.

Additional Considerations

The library aims to extend further, potentially including functions like trigonometric and logarithmic calculations adapted for fixed-point types.
It is crucial to emphasize that operations on Q and Sq types, which are fundamentally compile-time constructs built around basic integral types stored in memory at runtime, are not all thread-safe.

Example

Here's a simple example of using the library:

```cpp
#include <fpm.hpp>
using namespace fpm::types;

int main() {
    using i32q12_t = i32q12<-100.,  200.>;  // int32_t, q12
    using i32q14_t = i32q14<-100.,  200.>;  // int32_t, q14
    using u32q11_t = u32q11<   0., 1111.>;  // uint32_t, q11

    // Type resolutions
    // i32q12_t::res: 0.000244141, i32q14_t::res: 6.10352e-05,
    // u32q11_t::res: 0.000488281
    std::cout << "i32q12_t::res: " << i32q12_t::resolution << ", "
              << "i32q14_t::res: " << i32q14_t::resolution << ", "
              << "u32q11_t::res: " << u32q11_t::resolution << std::endl;

    auto a = i32q12_t::fromReal<105.45>();  // scaled to 431923
    i32q14_t b = 123.456_i32q14;  // via literal; scaled to 2022703
    i32q14_t a2 = a;  // copy-upscale from q12 to q14; scaled to 1727692

    /* do some math: math results have static-Q (Sq) types */

    // addition
    auto c = a + b;  // i32sq14<-200.,400.>; scaled: 3750395, real: 228.906

    // clamp value to narrower type
    auto d = i32q12_t::fromSq<Ovf::clamp>(c);  // scaled: 819200, real: 200.0

    // cast to unsigned type
    auto s = static_q_cast<u32q11_t, Ovf::unchecked>(d);  // sc: 409600, re: 200

    // multiplication
    auto p = a2 * d;  // i32sq14<-20000.,40000.>; scaled: 345538400, real: 21090

    // Sum: 200, Product: 21090
    std::cout << "Sum: " << s.real() << ", Product: " << p.real() << std::endl;
    // Sum[u32q11]: 409600, Product[i32sq14]: 345538400
    std::cout << "Sum[u32q11]: " << s.scaled() << ", "
              << "Product[i32sq14]: " << p.scaled() << std::endl;

    return 0;
}
```

For a more complex example of a computation, please refer to Practical Example.

Getting Started

To begin using the Fixed-Point Math Library in your project, start by including the library header file from the *inc* directory in your source code (make sure that the *inc* directory is known to the compiler):

```
#include <fpm.hpp>
```

Next, make use of the library's definitions and types by adding the following namespace directives to your source files:

```
using namespace fpm::types;
using Ovf = fpm::Ovf;  // optional
```

This setup will allow you to easily access and utilize the various fixed-point types provided by the library in your applications. For clarity, some examples in this documentation may include (parts of) these lines of prefix explicitly. Moving forward, these lines will not be shown in any further examples within this documentation.

Utilities

Scaling

One of the foundational declarations is the definition of `scaling_t` as an alias for `int`. The decision to make `scaling_t` signed is strategic because the framework needs to support both positive and negative fractional values:

- **Positive Scaling Values**: These are used to represent fractional parts, where positive scaling factors enhance the precision of the fractional component by specifying how many bits are dedicated to values below the integer point.
- **Negative Scaling Values**: Negative scaling is used to represent larger numbers than those typically representable with the given bits of the used integral base type. By scaling the real number by 2^f, where f is negative, large numbers can be represented at the cost of reducing the precision of the integral part of the number by $2^{|f|}$, effectively balancing the range and precision according to specific needs.

This flexibility ensures that the framework can adapt to a variety of numerical ranges and precision requirements, making it highly versatile for different applications.

Helper Functions

The library includes a set of helper functions designed to facilitate the manipulation of values at compile-time, ensuring that they are correctly scaled for use with different Q types. These functions are critical for creating and converting scaled values accurately and efficiently.

- **v2s (Value to Scale)**: This function is used to convert a real floating-point value into a scaled integer value at compile time. It multiplies the given `ValueT` value by 2^{to}, allowing for precise initialization of fixed-point variables from real numbers within the resolution of `TargetT`.
- **s2s (Scale to Scale)**: This function changes the scaling of a given scaled integer value. It multiplies the value by $2^{to-from}$, allowing conversion from one fixed-point scale to another without losing precision relative to the scaling factor. The function is used for rescalings in the internal implementations of `Sq` and `Q`. It proxies one of two implementations: `s2smd`, which uses multiplication and division for conversion, and `s2sh`, which uses binary shift operations. The latter is the default, while the former can be selected by defining the `FPM_USE_MULTDIV_TO_RESCALE` macro before including `fpm.hpp`.
- **scaled**: Converts a given real value (double) to its scaled representation, using the specified scaling factor `f`. This function is effectively an alias for `v2s<f, base_t>(real)`, streamlining the process of initializing `Q` types with precise scaled values derived from real numbers.
- **real**: Converts a scaled integral value back to its real (double) representation using the given scaling factor `f`. This function serves as an alias for `s2smd<f, 0, double>(scaled)`, facilitating the conversion of scaled values into their real, human-readable forms, particularly useful for outputs and debugging.

The functions are declared as follows:

```
namespace fpm {

template< scaling_t to, typename TargetT, /* deduced: */ typename ValueT >
consteval TargetT v2s(ValueT value) noexcept { /* ... */ }

template< scaling_t from, scaling_t to, std::integral TargetT >
constexpr TargetT s2s(std::integral auto value) noexcept { /* ... */ }

template< scaling_t f, std::integral TargetT = int >
consteval TargetT scaled(double real) noexcept { /* ... */ }

template< scaling_t f, typename TargetT = double >
constexpr TargetT real(std::integral auto scaled) noexcept { /* ... */ }

}
```

Here are practical examples demonstrating how to use the helper functions provided by the library:

- Using `fpm::v2s` to convert a floating-point number to a scaled integer value at compile-time:

```
// Converts 10.5 to a scaled integer using a scale factor of 2^3
auto scaledValue = fpm::v2s<3, int8_t>(10.5);  // 84: int8_t
```

- Using `fpm::s2s` to change the scaling of an already scaled integer value:

```
// Converts a value scaled by 2^8 to a scaling of 2^4
auto rescaledValue = fpm::s2s<8, 4, uint16_t>(256u);  // 16: uint16_t
```

- Using `fpm::scaled` to convert a real number to a scaled integer representation at compile-time:

```
// Converts 25.5 to a scaled integer with a scaling factor of 2^10
auto scaledValue = fpm::scaled<10, int16_t>(25.5);  // 26112: int16_t
int scaledValueInt = fpm::scaled<10>(25.5);  // 26112: int (default type)
```

- Using `fpm::real` to convert a scaled integer value back to its real number representation:

```
// Converts the scaled value 26112 back to its real value, scaling f=10
double realValue = fpm::real<10>(26112);  // 25.5: double (default type)
auto realValueInt = fpm::real<10, int>(26112);  // 25: int
```

These examples illustrate the utility of `v2s`, `s2s`, `scaled`, and `real` in fixed-point calculations, enabling crucial scaling adjustments for maintaining precision in embedded systems programming. These functions simplify the conversion between real numbers and scaled representations, enhancing both accuracy and efficiency.

Static Assertion

The library defines four compile-time-only static assertion functions to ensure that `Q` or `Sq` types conform to expected specifications:

- **static_assert_base**: This function statically asserts whether the `base_t` of a given `Q` or `Sq` type matches the specified base type `Base`.

```
// definition in fpm:
template< std::integral Base, class QSq >
consteval void static_assert_base() noexcept { /* ... */ }

// use as follows:
using someType = u8q4<1., 10.>;
auto someVariable = someType::fromReal< 5.5 >();

fpm::static_assert_base< uint8_t, someType >();
fpm::static_assert_base< uint8_t, decltype(someVariable) >();
```

- **static_assert_scale**: This function statically asserts that the scaling factor `f` of a given `Q` or `Sq` type is as specified.

```
// definition in fpm:
template< scaling_t f, class QSq >
consteval void static_assert_scale() noexcept { /* ... */ }

// use as follows:
using someType = u8q4<1., 10.>;
auto someVariable = someType::fromReal< 5.5 >();

fpm::static_assert_scale< 4, someType >();
fpm::static_assert_scale< 4, decltype(someVariable) >();
```

- **static_assert_limits**: This function statically asserts that the real value range of a given `Q` or `Sq` type falls within the specified `min` and `max` values.

```
// definition in fpm:
template< double min, double max, class QSq >
consteval void static_assert_limits() noexcept { /* ... */ }

// use as follows:
using someType = u8q4<1., 10.>;
auto someVariable = someType::fromReal< 5.5 >();

fpm::static_assert_limits< 1., 10., someType >();
fpm::static_assert_limits< 1., 10., decltype(someVariable) >();
```

- **static_assert_specs**: This comprehensive function statically asserts all critical properties (`base_t`, `f`, `min` and `max`) of a given `Q` or `Sq` type.

```
// definition in fpm:
template< std::integral Base, scaling_t f, double min, double max, class QSq >
consteval void static_assert_specs() noexcept { /* ... */ }

// use as follows:
using someType = u8q4<1., 10.>;
auto someVariable = someType::fromReal< 5.5 >();

fpm::static_assert_specs< uint8_t, 4, 1., 10., someType >();
fpm::static_assert_specs< uint8_t, 4, 1., 10., decltype(someVariable) >();
```

These assertions are especially useful for verifying the type of intermediate results after calculations with `Sq` types. If the core types of an application are changed, these assertions might trigger a compilation error, even if the formulas and calculations still compile. This ensures that certain type properties remain consistent throughout the development process, and any changes to the underlying type definitions are immediately flagged.

Q-Type

Q-Type

> *Just like the enigmatic Q from Star Trek, the* `Q` *type in the Fixed-Point Math Library possesses its own set of powerful capabilities, albeit more grounded in the realm of arithmetic precision than in galactic mischief.* -- ChatGPT

The primary purpose of `Q` is to efficiently implement a fixed-point type that mimics real, comma-separated numbers within a specific real value range, wrapping around a scaled integer value that can be stored and modified at runtime, ensuring dynamic and precise arithmetic operations. Additionally, `Q` provides robust overflow protection and facilitates seamless conversions to and from other `Q` types.

Template

`Q` is declared by the following template:

```
namespace fpm {

template< std::integral base_t,
          scaling_t f,
          double realMin,
          double realMax,
          Overflow ovfBx >
class Q {
    public:
    // ...
    private:
        base_t value;  ///< actual integer value stored in runtime memory
};

}
```

- **base_t**: Refers to the underlying integral type that holds the scaled value. Within the `Q` class, the scaled value, which represents the fixed-point number, is declared as a private member. This value is the only element that is stored in runtime memory for the `Q` instance.
- **f**: The number of fractional bits. This parameter, defined as `fpm::scaling_t`, determines the scaling factor, affecting how much of the integer's precision is used for fractional parts.
- **realMin** and **realMax**: These specify the minimum and maximum values of the real value range that `Q` can represent, effectively setting the bounds for compile-time computation. They are optional; if not explicitly set, the default is the largest feasible value range. For symmetric ranges in signed types, the default excludes the

minimum value of the base type to prevent potential overflows when taking the absolute value, thereby ensuring safe operations within predictable limits. For a `Q` type with a signed base type, if the value range includes only negative values up to and including zero, it is highly recommended to use `-0.0` (note the explicit minus sign) as the upper limit to clearly define the endpoint of the negative value range. This distinction is recommended because, although the runtime scaled integer does not differentiate between `-0` and `+0`, the `Q` type does at compile-time. This can make a difference in how the compiler interprets and handles the boundary conditions and overflow checks.

- **ovfBx**: Dictates the overflow behavior for operations that exceed the designated value range. The default overflow behavior is `Ovf::error`, which means the code will not compile if a potential overflow scenario is detected. This is particularly common when a conversion from a wider to a narrower real value range is attempted, safeguarding against inadvertent data loss or corruption.

These parameters are integral to the structure of each `Q` type and are set as static constexpr members, allowing them to be accessed directly for various computations or validations.

Additionally, a `Q` type provides the following static constexpr members to aid in precise and effective data manipulation:

- **scaledMin**: This member stores the minimum value that can be represented in the scaled format, calculated based on the *realMin* and the scaling factor.
- **scaledMax**: This member stores the maximum value that can be represented in the scaled format, calculated based on the *realMax* and the scaling factor.
- **resolution**: This represents the real resolution (double) of the `Q` type, defined as 2^{-f}. It indicates the smallest difference between two representable real values within the type, providing clarity on the granularity and precision at which values can be manipulated or interpreted.

Type Examples

Here are some examples of how `Q` types can be defined:

- `fpm::Q<int8_t, 4, -10., 10.>` represents a `Q8.4` fixed-point type, covering a real value range from -10 to +10. The default overflow behavior is set to `Ovf::error`, meaning any operation leading to overflow will not compile.
- `fpm::Q<uint32_t, 12, 0., 1000., Ovf::clamp>` defines a `UQ32.12` fixed-point type with a real value range from 0 to 1000. Overflow behavior is set to `Ovf::clamp`, meaning values that exceed the range are adjusted to the nearest boundary.

- `fpm::Q<in16_t, 8>` defines a `Q16.8` fixed-point type that uses `Ovf::error` as the default overflow behavior and the largest possible **symmetric** value range. This symmetric range excludes `INT16_MIN` as scaled value to avoid issues such as overflow when taking the absolute value.

Type Aliases

The library provides alias templates as syntax sugar for common `Q` types, looking similarly to standard integer types, in the namespace `fpm::types`. This simplifies the usage and readability of the types in your code. For instance:

- **`i32q16< realMin, realMax, Ovf::clamp >`** is a `Q<int32_t, 16, realMin, realMax, Ovf::clamp>`
- **`u16q8< realMin, realMax >`** is a `Q<uint16_t, 8, realMin, realMax>`
- **`i32qm7< realMin, realMax >`** is a `Q<int32_t, -7, realMin, realMax`
- **`i16q6<>`** is a `Q<int16_t, 6>`, with the largest possible symmetric value range and `Ovf::error`

Literals for Q Type Aliases

The `fpm` library automatically provides literals for all fixed-point types defined in the `fpm::types` namespace, ensuring a consistent and easy way to work with these types directly in your code. Each type has a corresponding literal suffix that mirrors its type declaration, enhancing code readability and precision.

An important characteristic of these literals is that the value range specified by the literal is only the value itself, which provides flexibility and avoids unnecessary limitation of calculations.

Here are examples of how these literals can be utilized:

```
#include <fpm.hpp>
using namespace fpm::types;

i32q8<-1000., 1000.> distance = -150.0_i32q8;
u16q4<0., 2000.> pressure = 1023.0_u16q4;
u8q4<0., 15.> smallValue = 14.5_u8q4;
```

These literals, named after their respective types (such as `_i32q8` for `i32q8` and `_u16q4` for `u16q4`), make it clear what type of fixed-point precision is being applied, making the code self-documenting and easier to understand. This standardized approach ensures that developers can immediately recognize the data types and precision being used just by looking at the literal suffix.

User Literals

C++ provides the flexibility to define user-defined literals, enhancing the expressiveness and clarity of code by allowing types to have their own suffixes. The `fpm` library supports this feature with a macro that simplifies the creation of these literals for specific `Q` types.

The macro `FPM_Q_BIND_LITERAL(_q, _literal)` enables the binding of a custom suffix to a `Q` type, making the use of fixed-point types more intuitive. For example:

```
// Define speed as a range from 0 to 100 meters per second.
using speed_t = i32q8<0., 100.>;
// Bind the 'm_per_s' literal to the speed_t type.
FPM_Q_BIND_LITERAL(speed_t, m_per_s)  // m/s

// Define cruising speed using the user-defined literal.
auto cruisingSpeed = 55.0_m_per_s;
```

Note: When using user-defined literals, a `_` must be placed between the number and the literal suffix, as shown in `55.0_m_per_s`. This ensures proper parsing and application of the literal in C++.

This approach not only makes the code more readable but also reduces the risk of errors by enforcing type-specific operations directly through the literal notation.

While user-defined literals offer a significant advantage in code clarity and type safety, there is a limitation to their usage. Each literal can only be bound once per compilation unit to a specific `Q` type. This restriction ensures that literals are consistently associated with their respective types throughout a single file, preventing conflicts and ambiguities.

Nevertheless, it is often practical for certain literals to be used across the entire application. To accommodate this, such literals should be defined in a core header file. This centralized approach allows for the universal application of the literal across multiple files, ensuring consistent usage and behavior throughout the application.

Overflow Behaviors

The `Q` type provides a robust handling of overflow scenarios through different behaviors that can be specified during the type definition. Understanding and selecting the appropriate overflow behavior is critical to ensure that your application handles edge cases in a predictable and controlled manner. The library currently implements four types of overflow behaviors (from strictest to most lenient):

- **Error (`Ovf::error`)**: This behavior causes a compiler error if overflow is possible. It is the strictest behavior and the **default setting**, ensuring that potential overflow scenarios are addressed during development rather than at runtime.
- **Assert (`Ovf::assert`)**: This overflow behavior triggers a runtime assertion and calls the application-defined `ovfAssertTrap()` function if overflow occurs. This is useful for debugging and development phases where catching errors immediately is crucial.
- **Clamp (`Ovf::clamp`)**: With this behavior, values that would normally overflow are clamped at runtime to the maximum or minimum value within the range defined for the `Q` type. This prevents overflow while still allowing the application to continue running.
- **Unchecked (`Ovf::unchecked`), Allowed (`Ovf::allowed`)**: This setting disables overflow checking entirely. It allows the value to wrap around according to the standard behavior of the underlying data type. "Unchecked" and "Allowed" refer to the same behavior, however, the choice of term may fit better semantically depending on the context or usage within specific parts of your application.

Choosing the right overflow behavior depends on your application's requirements for safety, debugging, and performance. It may even vary between different builds if corresponding declarations are used.

Construction

The `Q` class can be instantiated using various integral types and scaling factors to fit specific application requirements.

`Q` cannot be constructed directly from floating-point values at runtime to maintain type safety and prevent inadvertent data loss or misinterpretations. Instead, `Q` provides static constructor methods to create instances from real values, ensuring conversions are explicit and controlled:

- To construct a `Q` instance from a real constexpr value, you can use the static `fromReal< real >()` constructor function, which converts a floating-point number to the corresponding fixed-point representation. For example:

  ```
  using fpm::Q;

  // Construct from real value
  auto a = Q<uint16_t, 8, 100., 200.>::fromReal< 155. >();
  ```

  The `auto` keyword is very handy here because it spares us from needing to specify the type twice, as the compiler deduces the type at compile time.
- Additionally, Q supports construction from scaled constexpr integer values through the static `fromScaled< scaled >()` constructor function, which takes an integer already scaled to the fixed-point format:

  ```
  // Construct from scaled value
  auto b = Q<uint32_t, 8, 1000., 2000.>::fromScaled< 307200 >();
  auto c = Q<int32_t, 18, -10., 15.>::fromScaled< fpm::scaled<18>(12.) >();
  ```

- For creating instances of `Q` types with scaled integral runtime values, the constructor function `construct<ovfBxOvrd>( scaled )` is essential. It conducts a runtime overflow check as desired, adhering to the overflow behavior of the `Q` type, or, if specified, the behavior override provided via the template parameter `ovfBxOvrd`. This function ensures that values are instantiated safely, respecting the defined overflow strategies to prevent data corruption or unexpected behavior. It's important to note that this method can only be used with `scaled` integer values, as the runtime construction from double values is explicitly prohibited by the library's design to maintain type safety (no floats at runtime).

```
// Construct with runtime scaled value, clamped to target value range
int32_t scaled = 123456;  // scaled value, q16; real=1.88379
auto qVal = Q<int32_t, 16, -100., 100.>::construct<Ovf::clamp>(scaled);

// Overflow override not necessary if target type inherently clamps
auto qVal2 = Q<int32_t, 16, -100., 100., Ovf::clamp>::construct(scaled);
```

These construction methods provide precise control over how values are initialized in the `Q` type, ensuring adherence to the defined numerical ranges and overflow behaviors. Values are defined within the `< >` brackets at compile-time, and within the `( )` at runtime. It's important not to forget the `()` at the end of the static functions `Q<...>::fromReal<.>()` and `Q<...>::fromScaled<.>()`, as functions require parentheses to invoke them.

Type Aliases

Construction of `Q` instances using type aliases looks as follows:

```
using namespace fpm::types;  // predefined type alias templates

// user-defined type aliases with specific value ranges
using i32qm7_t = i32qm7<4e10, 5e10, Ovf::clamp>;
using i16q6_t = i16q6<>;

auto a = u16q8<100., 200.>::fromReal< 155. >();
auto c = i32q18<1000., 1500.>::fromScaled< 314572800 >();
auto d = i32qm7_t::fromReal< 4.567e10 >();
auto e = i16q6_t::fromReal< 444.4 >();

i32qm7_t f = 6.2e10_i32qm7;  // clamped to 5e10
i16q6_t g = -333.33_i16q6;
```

In the examples provided in this document, these aliases will mostly be used for construction, as they offer a more concise and familiar notation.

Value Access

Within the `Q` class, the actual scaled value that is stored in runtime memory can be accessed through specific member functions. These functions provide direct and controlled access to both the scaled representation and the real-value representation of the `Q` type:

- **Accessing the Scaled Value:** The scaled value can be accessed using the `scaled()` member function. This function returns the value in its internal scaled format, which is useful for low-level operations or when interfacing with systems that require the scaled integer directly.

```
auto scaledVal = qValue.scaled();  // Access the internal scaled value
```

- **Accessing the Real Value:** To obtain the real value as a floating-point number, you can use the `real()` function. This method converts the value back to its double precision floating-point representation, primarily useful for debugging reasons. It's important to use this function cautiously due to potential floating-point inaccuracies and the computational overhead associated with floating-point operations.

```
double realVal = qValue.real();  // Get the real value as double
```

- **Accessing the Real Value as an Integer:** For accessing the real value in any integer format, use `real<typename>()`. This method converts the unscaled real value to the given integral type by truncating it according to C++ rules, discarding any fractional digits. This truncation must be handled with care to avoid unintended data loss, especially when the type has a large fractional part.

```
// Get the real value as an integer, truncating fractional digits
int intRealVal = qValue.real<int>();
```

Rescaling

In situations where the base type remains the same but the fractional part `f` differs between two `Q` types, rescaling is necessary. This process is referred to as "up-scaling" when `f` increases, and "down-scaling" when `f` decreases.

- **Up-Scaling**: Up-scaling is achieved by **multiplying** the source value by $2^{to-from}$, effectively increasing the number of fractional bits to fit the higher resolution of the target type. While this process is technically lossless because no data is discarded, it doesn't enhance the actual resolution or detail of the original data; the additional fractional bits are essentially filled with zeros.
- **Down-Scaling**: Down-scaling reduces the number of fractional bits by **dividing** the source value by $2^{from-to}$. This operation is inherently lossy as it involves discarding the least significant bits of the fractional part. Consequently, the precision of the original value is reduced to match the lower resolution of the target type, resulting in a loss of detail that reflects the decreased fractional granularity.

To rescale a `Q` type to another with the same base type but different `f`, you can use simple assignment, or, if necessary, the `QTo::fromQ<ovfBxOvrd>( qFrom )` constructor function, both of which handle the scaling adjustments internally:

```
auto sourceValue = i32q10<-500., 1500.>::fromReal< 1024. >();

// Rescale from f=10 to f=8 and a wider value range via assignment
i32q8<-510., 1500.> targetValue1 = sourceValue;  // automatically rescaled

// Rescale from f=10 to f=8 and clamp to the narrower value range via fromQ<>()
auto targetValue2 = i32q8<-450., 1450.>::fromQ<Ovf::clamp>( sourceValue );
```

Overflow checks during scaling adjustments are unnecessary if the real value range of the target type is the same as or wider than that of the source type, and if the overflow behavior of the target type is the same or less strict than that of the source type. Under these conditions, values can be directly assigned.

However, if the target value range is narrower than the source's, or if the target type implements a stricter overflow behavior than the source type, you may need to use the `fromQ<ovfBx>(.)` function to specify a different overflow behavior. Typically, the value is clamped to fit within the target type's range, ensuring correct handling of potential overflow situations without causing compilation errors or runtime assertions. This adjustment does not permanently change the overflow behavior of the target type but applies the specified overflow handling only for the conversion to the target type.

```
using source_t = i32q10<-500., 1500.>;   // i32q10, Ovf::error
using overflow_t = i32q10<-500., 1500., Ovf::allowed>;
using coarser_t = i32q8<-500., 1500.>;   // i32q8, same value range as source_t
using wider_t = i32q8<-600., 1600.>;     // i32q8, wider value range
using narrower_t = i32q8<-400., 1500.>;  // i32q8, narrower value range

auto source = source_t::fromReal< 1024.7 >();
auto ovfSrc = overflow_t::fromReal< -555.5 >();  // value out of range

/* same value range: direct assignment can be used if ovf behavior allows it */
coarser_t a1 = source;                          // ok
auto a2 = coarser_t::fromQ( source );           // also ok
// coarser_t a3 = ovfSrc;                       // error (stricter ovf behavior)
auto a4 = coarser_t::fromQ<Ovf::clamp>( ovfSrc );  // ok (runtime clamp needed)
i32q8<-500., 1500., Ovf::clamp> a5 = ovfSrc;  // ok (clamping target)

/* wider value range: direct assignment can be used if ovf behavior allows it */
wider_t b1 = source;                              // ok
auto b2 = wider_t::fromQ( source );               // also ok
// wider_t b3 = ovfSrc;                           // error (stricter ovf beh.)
auto b4 = wider_t::fromQ<Ovf::clamp>( ovfSrc );  // ok (runtime clamp needed)
i32q8<-600., 1600., Ovf::clamp> b5 = ovfSrc;     // ok (clamping target)

/* narrower range: fromQ<>() is required unless target type checks anyway */
// narrower_t c1 = source;  // error (narrower range)
auto c2 = narrower_t::fromQ<Ovf::clamp>( source );  // ok
auto c3 = narrower_t::fromQ<Ovf::clamp>( ovfSrc );  // ok
i32q8<-400., 1500., Ovf::clamp> c4 = ovfSrc;        // ok (clamping target)
i32q8<-400., 1500., Ovf::unchecked> c5 = ovfSrc;    // ok (check discarded)
```

Casting

In the `Q` type framework, casting between different `Q` types with varying base types necessitates explicit type conversion. The simplest form of casting involves using `static_cast< QTo >( qFrom )`, which is straightforward when scaling adjustments are feasible and the target value range is either the same or wider than the source's, and if the target type's overflow behavior is the same or less strict.

```
auto sourceValue = i32q10<0., 1500.>::fromReal< 1024. >();

// cast i32q10 to unsigned u32q12
auto cast1 = static_cast< u32q12<0., 1500.> >(sourceValue);

// cast i32q10 to i16q4, explicitly dropping any overflow check in target type
// (not recommended though, target type will drop any overflow check this way)
auto cast2 = static_cast< i16q4<0., 1030., Ovf::unchecked> >(sourceValue);
```

However, if the target range is narrower and the target type incorporates overflow checks, or if the target type's overflow behavior is more restrictive than that of the source, `static_cast` may not be sufficient. In cases where overflow could occur, and unless the target type handles overflow with `Ovf::clamp` or `Ovf::assert` when constructed, a simple `static_cast` will not properly address potential overflow issues and thus not compile.

To manage potential overflow effectively, `Q` provides three specialized casting functions:

- `static_q_cast< QTo, ovfBxOvrd >( qFrom )`: This function applies the overflow handling as specified by the target type or uses the overridden overflow behavior if provided. It ensures that the conversion adheres to the designated overflow protocols, **performing runtime checks only when necessary** from a static point of view.
- `safe_q_cast< QTo, ovfBxOvrd >( qFrom )`: This function **performs a runtime overflow check regardless of whether it is strictly necessary**. It requires that the overflow behavior is set to a runtime check `Ovf::clamp` or `Ovf::assert`, and does not permit `Ovf::unchecked` and the compile-time check `Ovf::error`. This method is designed for scenarios where ensuring data integrity is critical, and overflow must be actively managed.

- `force_q_cast< QTo >( qFrom )`: This casting function **performs no checks and simply reinterprets the scaled source value**, constructing the specified `ToQ` type around it, potentially truncating bit information. This is essentially a forced value reinterpretation and should be used with caution, as it bypasses all overflow and range checks as well as rescaling. This method is suitable for scenarios where the developer is certain of the data integrity and the applicability of the conversion.

These casting functions provide robust tools for managing different fixed-point types, ensuring that conversions are both safe and efficient, depending on the operational requirements and data integrity needs.

```
auto value = i32q10<0., 1500.>::fromReal< 1234. >();

// cast value to unsigned u32q12 (identical to static_cast)
auto cast1 = static_q_cast< u32q12<0., 1500.> >( value );  // 1234.

// cast value to i16q4 with narrower range, overflow check is dropped explicitly
// --> value intentionally is out of range
auto cast2 = static_q_cast< i16q4<0., 1000.>, Ovf::allowed >( value );  // 1234.

// static-cast cast2 (potentially out of range) to a wider range
// note: no overflow checks are performed (wider range, identical ovf behavior)
auto c3static = static_q_cast< i16q4<-100., 1100.> >( cast2 );  // 1234.

// safe-cast of cast2 is needed to really catch and clamp the out-of-range value
auto c3safe = safe_q_cast< i16q4<-100., 1100.>, Ovf::clamp >( cast2 );  // 1100.

// force-cast cast2 (i16q4) into a u8q1
// --> u8q1<0., 100.>::fromScaled< cast2.scaled() % 256 >() = 32|scaled, 16|real
auto forced = force_q_cast< u8q1<0., 100.> >( cast2 );  // 16.
```

Range Clamping

The `Q` type framework includes a specialized feature known as "range clamping" to address the nuances of casting between differently signed types of different sizes. This feature ensures that conversions between signed and unsigned types adhere to intuitive expectations regarding value ranges:

- **From Signed to Unsigned**: When a `Q` type with a signed base type is cast to a `Q` type with an unsigned base type of the same or smaller size, any values in the negative range of the signed type are clamped to the lower limit of the unsigned type. This clamping is crucial to avoid underflow and to ensure values remain within the legitimate range of the unsigned type.
- **From Unsigned to Signed**: Similarly, when a `Q` type with an unsigned base type is cast to a `Q` type with a signed base type of the same or smaller size, values that fall

in the upper half of the unsigned range are clamped to the upper limit of the signed type. This measure prevents potential overflow, ensuring that the values do not exceed the maximum positive value of the signed type.

- **From Larger Signed Type to Smaller Signed Type**: When casting from a larger signed `Q` type to a smaller signed type, clamping adjusts any values that exceed the storage capabilities of the smaller type to the nearest boundary. This is crucial for preventing data overflow or underflow, helping maintain the integrity of the data.
- **From Larger Unsigned Type to Smaller Unsigned Type**: In cases where a larger unsigned `Q` type is cast to a smaller unsigned type, the values that exceed the maximum representable value of the smaller type are clamped to this maximum. This adaptation is essential for preserving data correctness by preventing overflow.

```
auto iN = i32q10<-1300., 1300.>::fromReal< -1234. >();
auto iP = i32q10<-1300., 1300.>::fromReal< 1234. >();
auto uP = u16q8<0., 255.>::fromReal< 234. >();

// cast negative signed i32q10 to unsigned u32q5
auto iN2u = static_q_cast< u32q5<0., 3000.>, Ovf::clamp >( iN );  // 0.

// cast positive signed i32q10 to smaller unsigned u16q8
auto iP2u = static_q_cast< u16q8<0., 255.>, Ovf::clamp >( iP );  // 255.

// cast negative signed i32q10 to signed i16q2 with narrower value range
auto iN2i = static_q_cast< i16q2<-1000., -0.>, Ovf::clamp >( iN );  // -1000.

// cast positive signed i32q10 to signed i8q1
auto iP2i = static_q_cast< i8q1<-60., 60.>, Ovf::clamp >( iP );  // 60.

// cast unsigned u16q8 to signed i8q1
auto u2i = static_q_cast< i8q1<-60., 60.>, Ovf::clamp >( uP );  // 60.

// cast unsigned u16q8 to unsigned u8q2 with narrower value range
auto u2u = static_q_cast< u8q2<0., 30.>, Ovf::clamp >( uP );  // 30.
```

These adjustments are driven by internal logic designed to mirror what would typically be expected from the real values during such type conversions. By incorporating this feature, the framework helps preserve data integrity and provides a logical, safe transition between signed and unsigned `Q` types, reflecting the expected behavior of real-world values.

How Does Range Clamping Work?

The process of range clamping in the `Q` type framework is designed to handle conversions between types of different sizes and signs meticulously to avoid data loss and ensure integrity. Here's a detailed breakdown of how this process works:

1. **Rescaling the Base Value**: Initially, the base value of the source `Q` type is rescaled in a `scale_t`, which has at least twice the size of the target type but is not smaller than the size of the source type, and retains the same sign as the source `Q` type. This step is crucial as it preserves the sign bit during the scaling process, particularly important when the source type is signed and the operation might involve a reduction in size (down-cast).
2. **Casting to a Transitional Type**: The scaled value is then cast to a transitional `cast_t`, which has twice the size and the sign of the target `Q` type. This intermediary step ensures that the value can be adjusted in a way that conforms to the sign characteristics of the target type, facilitating a smoother transition and accurate clamping in subsequent steps.
3. **Clamping and Final Adjustment**: In the final stage, the value now represented in `cast_t` allows for proper clamping. For instance, if the original value was negative and the target type is unsigned, the value falls within the upper half of the value range of `cast_t`. This positioning is key as it indicates that the value requires clamping to the lower limit of the unsigned target type to be clamped intuitively. Conversely, if the original value was positive and fits into the value range of the target type, the cast value will fit and not be clamped. The larger intermediate type (`cast_t`) provides the necessary headroom to accommodate these adjustments without overflow or underflow, ensuring that the final value is accurate and within the expected range.

Sq-Type

Sq-Type

The `Sq` type, standing for "Static Q," is a specialized variant of the `Q` type designed to handle constant values at runtime. Unlike the `Q` type, which accommodates dynamically changing values, the `Sq` type maintains a fixed value throughout its lifetime, making it particularly useful in scenarios where stability and predictability are paramount.

One of the principal advantages of the `Sq` type is the elimination of runtime overflow checks during computations. Because `Sq` types hold static values, the compiler can perform all necessary checks at compile time, ensuring that any formulas involving `Sq` types are verified for validity within the predefined value ranges. This static analysis significantly enhances performance and reliability, as it prevents runtime errors related to overflow and underflow.

`Sq` types are equipped with a range of mathematical functions and arithmetic operators, allowing them to efficiently handle a variety of calculations and operations that are integral to their use cases. This feature enables `Sq` types to function almost autonomously within their defined scope, facilitating complex mathematical operations without the need for runtime value adjustments.

However, while `Sq` types offer robust safety during calculations, they require careful management during type conversion. Overflow checks become necessary when converting between `Q` and `Sq` types, specifically at the points of conversion both to and from `Sq`. These checks are critical to ensure that the static values adhered to by `Sq` types are compatible with the potentially dynamic values of `Q` types.

In terms of scope and lifecycle, `Sq` types are inherently limited to the block of code in which they are defined. They cannot naturally persist or move beyond this local scope without conversion to a `Q` type. This characteristic makes `Sq` types ideally suited for temporary computations and intermediate steps within complex algorithms where values do not need to be retained post-computation.

Conversely, `Q` types are designed to store and manage runtime values that may need to be maintained or modified over time and across different scopes. The flexibility of `Q` types makes them indispensable for applications that require the storage and manipulation of values beyond a fixed context, providing a bridge from the localized, static environment of `Sq` types to broader, dynamic applications.

In summary, while both `Q` and `Sq` types are integral to the Fixed-Point Math Library, each serves distinct roles: `Sq` types for static, localized computations within a confined scope, and `Q` types for dynamic, persistent data management across wider operational contexts.

Template

Similar to the `Q` type, the `Sq` type is defined using a template that specifies several critical parameters, enabling it to accurately represent fixed-point numbers within a specific real value range. The template for declaring an `Sq` type is outlined below:

```
namespace fpm {

template< std::integral base_t,
          scaling_t f,
          double realMin,
          double realMax >
class Sq {
public:
    // ...
private:
    base_t const value;  ///< Constant value, ensuring the static nature of Sq
};

}
```

The parameters for the `Sq` type template are as follows:

- **base_t**: The base data type, which must be an integral type. This defines the type of the scaled integer that `Sq` uses to store its fixed-point number.
- **f**: The number of fractional bits. This parameter, defined as `fpm::scaling_t`, determines the scaling factor, affecting how much of the integer's precision is used for fractional parts.
- **realMin** and **realMax**: These parameters specify the minimum and maximum real values that the `Sq` type can represent. They are optional; if not explicitly set, the default is the largest feasible value range. For symmetric ranges in signed types, the default excludes the minimum value of the base type to prevent potential overflows when taking the absolute value, thereby ensuring safe operations within predictable limits. For signed base types where the range includes only negative values up to and including zero, it is highly recommended to use `-0.0` (note the explicit minus sign) as the upper limit to ensure clarity and prevent ambiguities at the zero boundary.

It's important to note that **the `Sq` type does not implement any overflow behavior**, reflecting its purpose for handling ranges of values that are constant and known at compile-time, thereby eliminating the need for runtime overflow checks.

Additionally, the `Sq` type provides the following static constexpr members:

- **scaledMin** and **scaledMax**: These members represent the minimum and maximum scaled integer values that can be derived from the specified `realMin` and `realMax`. These values ensure that all operations stay within the bounds defined by the fixed-point format of the `Sq` type.
- **resolution**: This member indicates the smallest increment in the real value range of the `Sq` type, calculated as 2^{-f}. It specifies the precision level of the `Sq`, providing clear information about the smallest change in value that the type can represent, essential for high-precision calculations.

Type Examples

To illustrate the versatility and utility of the `Sq` type, here are a few examples showing how different configurations can be applied to accommodate various numeric ranges and precisions:

- `fpm::Sq<int16_t, 8, -100., 100>` represents an `Sq` type that uses a 16-bit signed integer (`int16_t`). The number of fractional bits is 8 (`f=8`), allowing for a decimal precision that suits a real value range of -100 to +100.
- `fpm::Sq<uint32_t, 12>` utilizes a 32-bit unsigned integer (`uint32_t`) with 12 fractional bits (`f=12`) and a full real value range, which is determined based on the capabilities of the unsigned integer and the specified number of fractional bits.

These examples demonstrate the flexibility of the `Sq` type in adapting to various needs by simply adjusting the base type, the number of fractional bits, and optionally (yet recommended), the value range.

Aliases and Literals

To simplify the usage of common `Sq` types in the library, shorter type aliases are provided via the `fpm::types` namespace. These aliases allow for more concise and readable code, particularly when dealing with commonly used configurations. Additionally, literals are defined for each alias to facilitate straightforward and error-free value assignments directly in the code.

Aliases for Common Types

Aliases exist for the most common `Sq` (and `Q`) types, covering all integral types up to 32 bits and scalings `f` ranging from $-bitnum/2$ to $bitnum - 1$.
Here are examples of some type aliases for `Sq` types:

- **`i32sq16< realMin, realMax >`**: Alias for `fpm::Sq<int32_t, 16, realMin, realMax>`. This configuration uses a 32-bit signed integer with 16 fractional bits, suitable for a real value range defined by `realMin` and `realMax`.
- **`u16sq7< realMin, realMax >`**: Alias for `fpm::Sq<uint16_t, 7, realMin, realMax>`. This version employs a 16-bit unsigned integer with 7 fractional bits, accommodating the specified real value range.
- **`i8qm2<>`**: Alias for `fpm::Sq<int8_t, -2>`. This configuration uses an 8-bit integer with `-2` fractional bits, effectively scaling the range upwards to accommodate larger values. The symmetric real value range achievable with this setup is from `-508.` to `508.`, facilitated by the negative scaling which expands the representable value range at the cost of precision (2^2).

Literals for Sq Type Aliases

To enhance usability, literals corresponding to each type alias are provided, allowing for immediate and clear value initialization:

- `-45.4_i32sq16` results in an `i32sq16<-45.4, -45.4>` initialized with a real value of `-45.4`.
- `15.5_u16sq7` creates an `u16sq7<15.5, 15.5>` with the real value `15.5`.
- `256_i8sqm2` creates an `i8sqm2<256, 256>` with a real value of `256`.

It's crucial to include an underscore between the numeric value and the literal suffix, as in `15.5_u16sq7` and `-45.4_i32sq16`, to ensure correct parsing and association in the source code.

Why is the Value Range Restricted to the Literal Value?

When an `Sq` type (or `Q` type) is initialized from a literal, the value range is intentionally set to be exactly the same as the value. This approach ensures that when compile-time calculations are performed involving these `Sq` types' limits, the potential for value range expansions that could lead to overflow (which would not compile) is minimized. Essentially, by limiting the `Sq` type to the precise value given in the literal, the type becomes highly predictable and efficient in computations, particularly in environments where precise control over numerical limits and performance is critical. This strategy simplifies handling edge cases and ensures that calculations remain as streamlined and efficient as possible, especially when multiple `Sq` types interact within an expression or algorithm.

Construction

The `Sq` type allows for compile-time construction from scaled integers and real floating-point values. Unlike the `Q` type, `Sq` has a constant value at runtime from a value range known at compile-time and does not implement any overflow behavior.

Compile-Time Construction

From Scaled Integers

To construct an `Sq` instance from a scaled integer at compile-time, the constructor function `Sq<...>::fromScaled<.>()` is used. Here are some examples:

```
// Constructing i8sq2 with a value range from -30.0 to 30.0, and value 42
auto sq1 = fpm::Sq<int8_t, 2, -30., 30.>::fromScaled< 42 >();  // real: 10.5

// Constructing u32sq16 with a value range from 0.0 to 1000.0, and value 65536
auto sq2 = u32sq16<0., 1000.>::fromScaled< 65536 >();  // real: 1.0

// Constructing i32sq8 with a full symmetric value range, and a value 256
auto sq3 = fpm::Sq<int32_t, 8>::fromScaled< 128 >();  // real: 0.5
```

The template parameters specify the base type, the number of fractional bits and optionally the real value range of the desired `Sq` type. The static `fromScaled` constructor function initializes the `Sq` instance with the provided scaled integer value. This value is stored at runtime if the `Sq` variable is used at runtime. When specifying the value, it should be enclosed within the angle brackets `< >`. Additionally, the parentheses `()` at the end of the constructor function invocation should not be forgotten, as they are needed to invoke the `fromScaled` function.

From Real Values

Similarly, `Sq` instances can be constructed from real floating-point values at compile-time using the `Sq<...>::fromReal<.>()` constructor function:

```
// Constructs i16sq4 with a value range from -100. to 100., from real value 6.3
auto sqR1 = fpm::Sq<int16_t, 4, -100., 100.>::fromReal< 6.3 >();  // scaled: 100

// Constructs i8sq1 with a value range from -50. to 50. and a real value 35.67
auto sqR2 = i8sq1<-50., 50.>::fromReal< 35.67 >();  // scaled: 71
```

From Literals

Literals can be utilized to construct `Sq` variables through explicit assignment, where an *rvalue* created from a literal can be assigned to an *lvalue* variable that has the same or a wider real value range. If the literal value is out of range, the expression does not compile. This feature increases readability while still ensuring type safety. Here are some examples demonstrating this approach:

```
// Constructing Sq from literal with single value within the variable's range
u16sq7<0., 200.> sqLit1 = 89.32_u16sq7;

// Constructing Sq from literal without range specification in type alias,
// assuming full range
i32sq20<> sqLit2 = -444.56_i32sq20;

// Does not compile: valid literal, however out of variable range
// i8sq2<-20., 20.> sqLit3 = 30.5_i8sq2;
```

Runtime Construction

Unlike the `Q` type, `Sq` does not have a runtime `construct<>()` method. Direct construction of `Sq` instances from runtime variables is not possible; instead, construction through a `Q` variable via the `qVar.toSq< realMin, realMax, ovfBxOvrd >()` method is necessary to enforce proper runtime overflow checks.
Each `Q` type inherently provides a corresponding `Sq` type with similar properties, which is accessible via `Q<...>::Sq<>`. Instances of this `Sq` type are constructed using the aforementioned `toSq()` method:

```
// Constructing Q instance from a runtime scaled integer
int32_t someVariable = fpm::scaled<16, int32_t>()
auto qV = i32q16<-10., 50.>::fromReal< 42.5 >();  // note: Q not Sq

// Constructing Sq instance from Q value; this Sq has similar properties than Q;
// Conversion is trivial, no overflow check needed
auto sqFromQ = qV.toSq();  // Sq<int32_t, 16, -10., 50.>, real value: 42.5

// toSq<>() can be used to clamp the Q value to a narrower Sq type
auto sqFromQ2 = qV.toSq<0., 38., Ovf::clamp>();  // i32sq16<0.,38.>, v: 38|real

// if the target Sq type has a wider range, no clamping is needed
auto SqFromQ3 = qV.toSq<-20., 50.>();  // i32sq16<-20.,50.>, real value: 42.5
```

Additionally, `Q` implements a unary `+` operator, which provides a shorthand for constructing a similar `Sq`-typed variable from a `Q` variable. This unary operator functions similarly to `qVar.toSq()`:

```
auto qVar = fpm::Q<int32_t, 16, -10., 50.>::fromReal< -8.8 >();

// conversion of a Q variable to an Sq variable with similar properties:
auto sqVar = +qVar;  // i32sq16<-10., 50.>, real value -8.8
```

Furthermore, when used in formulas, `Q` variables are implicitly converted to similar `Sq` variables. This implicit conversion feature is designed to streamline the syntax by reducing the clutter caused by explicit conversion overhead. For instance, in expressions involving multiple operations or different data types, `Q` variables can be seamlessly integrated without the need to manually convert each one, thus simplifying the code and enhancing its readability.

Value Access

The actual `Sq` value stored at runtime can be accessed through specific member functions. These functions provide controlled access to both the scaled representation and the real-value representation of the `Sq` type:

- **Accessing the Scaled Value:** The scaled value, which is the internally stored scaled integer, can be accessed using the `scaled()` member function. This is particularly useful for operations that require interaction with systems needing the integer directly or when performing low-level operations that depend on the scaled format.

  ```
  auto scaledVal = sqVar.scaled();  // Retrieve the internal scaled value
  ```

- **Accessing the Real Value:** To retrieve the real value as a floating-point number, the `real()` function is used. This method converts the internal scaled integer back to its double precision floating-point representation. It is primarily useful for applications that require precise numerical outputs, such as for display or in high-level calculations. However, it's important to handle this value carefully due to the potential for floating-point inaccuracies and the computational overhead associated with floating-point operations.

  ```
  double realVal = sqVar.real();  // Get the real value as double
  ```

  Similar to `Q`, the `real<int>()` variant can be used here to retrieve the integer part of the real value with the fractional part truncated. This truncation must be handled with care to avoid unintended data loss, especially when the type has a large fractional part.

Rescaling and Casting

Rescaling and casting of `Sq` variables, compared to their counterparts in `Q` types, are considerably more restricted. The design of `Sq` types emphasizes compile-time checks and static value management, which inherently limits dynamic operations such as rescaling and casting. Here are the key aspects of rescaling and casting for `Sq` variables:

Conditions for Rescaling: Rescaling is permitted only if the target `Sq` type has the same base type (`base_t`) and a different number of fractional bits (`f`). Importantly, the target type must also have a value range that is the same or wider than the source type to avoid potential overflows. The code does not compile if this is not the case. These measures ensure that the precision and range can be adjusted without altering the fundamental data type or exceeding the numerical capacity.

```
i16sq4<-100., 100.> sqSource = 25.75_i16sq4;  // scaled: 412

// Safe rescaling via assignment to a target type with the same base type
// and a compatible real value range
i16sq6<-100., 200.> sqRescaled = sqSource;  // scaled: 1648, real: 25.75

// Basically the same, but a bit more explicit.
auto sqRescaled2 = i16sq6<-100., 200.>::fromSq( sqSource );  // scaled: 1648
```

Restrictions on Casting: Casting between different base types requires an explicit `static_cast` to ensure that changes in data representation are handled correctly. The target type must have a value range that is the same or broader than the source type to ensure that no overflow errors occur during the casting process.

```
i16sq4<-60., 60.> sqI16 = -56.7_i16sq4;  // scaled: -907, real: -56.6875

// Explicit cast required to a target type with a different base type
auto sqI32 = static_cast< i32sq3<-60., 60.> >( sqI16 );  // s: -453, r: 56.625
auto sqI8 = static_cast< i8sqm4<-60., 60.> >( sqI32 );   // s: -3, r: -48
```

Note: In contrast to the `Q` type, there is no `safe_cast` and no `force_cast` for the `Sq` type. Both functions would contradict the foundational concept of `Sq`, which is designed to ensure compile-time safety and immutability of values, preventing any changes that could lead to runtime check overhead, errors, or value inconsistencies.

Compile-Time Overflow Checks: The `Sq` type enforces `Ovf::error` as its inherent - and only - overflow behavior. If the target type's value range is narrower than the source type's, the compiler will block the operation to prevent any risk of overflow, thus maintaining strict type safety.

```
i32sq4<-100., 100.> sqSource = -78.1_i32sq4;

// Error - target value range is narrower
i32sq5<-50., 50.> sqRescaled = sqSource;

// Error - target value range is narrower
auto sqCast = static_cast< i16sq4<-50., 50.> >( sqSource );
```

Using `Q` for Handling Overflows: Dynamic overflow handling, such as clamping, must be implemented through `Q` types since `Sq` does not support runtime overflow-checking behaviors. This approach is necessary when adjusting value ranges dynamically or when dealing with potential overflows.

```
i32sq4<-100., 100.> sqSource = -78.1_i32sq4;  // scaled: -1249

// Handling overflows by clamping via Q type; scaled: -960, real: -60
auto sqClamped = i32q4<-60., 60.>::fromSq< Ovf::clamp >( sqSource ).toSq();
```

This technique provides precise control over the timing of overflow checks within the `Sq` scope. By using `Q` for intermediate calculations, developers can ensure that overflows are addressed exactly when needed.

Additional Note: There are only a few edge cases where overflow handling on `Sq` through `Q` is even needed. These typically arise when the `Sq` variable is constructed from a variable of a `Q` type that uses explicit `Ovf::allowed`. In such cases, the potential for overflow exists because the `Q` type permits it, and the value, which potentially is out of range, can be passed to `Sq` via the `toSq()` method. Crucially, when such out-of-range values are passed, the compiler will still assume that the value is within the specified range of the `Sq` type. This might be desired behavior for some edge case computations, where handling or ignoring overflow could be beneficial. This is precisely why `Ovf::allowed` exists — to enable these specific scenarios where standard overflow management practices are intentionally bypassed.

Arithmetics

Overview

The primary capability of the `Sq` type is its extensive set of arithmetic operators and mathematical functions. These are designed to check the operations on a range of values at compile-time to assess potential overflow risks. If the evaluation confirms that the operations are safe within the value range of a specified `Sq` type, the code compiles. This compile-time safety check ensures that the operators and functions are reliable for use at runtime, enhancing the robustness of the `Sq` type.

Conversion of `Q` to `Sq`:
For variables of `Q` type, the `toSq()` method or the unary `+` operator can be utilized to transform them into a corresponding `Sq` type variable. This transformation ensures that dynamic values handled by the `Q` type are accurately converted into the static context of the `Sq` type, maintaining the integrity and constraints of the originally defined value range. If necessary, this conversion includes an overflow check, which by default adheres to the overflow behavior specified for the `Q` type. However, this behavior can be overridden when using the `toSq()` method. This ensures that any potential overflow issues are addressed during the conversion.

```
auto qVar = i32q10<-1000., 1000.>::fromReal< -555.55 >();
auto sqVar1 = qVar.toSq<-500., 500., Ovf::clamp>();  // clamps; real: -500.0
auto sqVar2 = +qVar;  // inherits the range -1000 to 1000; real: -555.55
```

Unary Operators:
`Sq` types support unary operators to manipulate the sign or value of an instance directly:

- **Unary Plus (+)**: This operator returns the value of the `Sq` instance as is, essentially a no-operation (no-op) in terms of value change.
- **Unary Minus (-)**: This operator returns the negated value of the `Sq` instance, effectively flipping its sign.

Binary Operators:
`Sq` types implement the standard arithmetic operators to perform calculations between instances or between an instance and a scalar (integral constant):

- **Addition (+)**: Adds two `Sq` instances, producing a new `Sq` instance with the resultant value.
- **Subtraction (-)**: Subtracts one `Sq` instance from another.

- **Multiplication (*)**: Multiplies two `Sq` instances or an `Sq` instance with a scalar.
- **Division (/)**: Divides one `Sq` instance by another or by a scalar.
- **Modulo (%)**: Computes the remainder of division between two `Sq` instances or an `Sq` instance and a scalar.

Comparison Operators:
Comparison operations are critical for logic and control flow in programming, and `Sq` types support all standard comparison operators:

- **Equal to (==)**
- **Not equal to (!=)**
- **Less than (<)**
- **Greater than (>)**
- **Less than or equal to (<=)**
- **Greater than or equal to (>=)**

These operators facilitate decisions and comparisons, ensuring that `Sq` instances can be directly used in conditional statements.

Shift Operators:
Shift operators adjust the bit representation of the fixed-point values:

- **Left Shift (<<)**: Shifts the bits of an `Sq` instance to the left, effectively multiplying the value by a power of two.
- **Right Shift (>>)**: Shifts the bits of an `Sq` instance to the right, effectively dividing the value by a power of two, rounded towards $-\infty$ to the nearest integer.

Clamping Functions:
Clamping functions are essential to maintain values within a specific range, both at runtime and compile-time, especially when dealing with edge cases:

- **Clamp to Minimum**: Ensures the `Sq` instance does not fall below a specified minimum value.
- **Clamp to Maximum**: Ensures the `Sq` instance does not exceed a specified maximum value.
- **Clamp to Range**: Restricts the `Sq` instance within a specified minimum and maximum range, combining both of the above functionalities.

Mathematical Functions:
The `Sq` type includes several built-in mathematical functions to extend its usability:

- **Absolute (abs)**: Returns the absolute value of an `Sq` instance.
- **Square (sqr) and Cube (cube)**: Functions to compute the square (x^2) and cube (x^3) of an `Sq` instance.
- **Square Root (sqrt) and Cube Root (cbrt)**: Functions to calculate the square root and cube root of an `Sq` instance.
- **Reverse Square Root (rsqrt)**: Calculates the reciprocal of the square root, commonly used in graphics and physics calculations to improve performance.
- **Minimum (min) and Maximum (max)**: Functions to determine the minimum and maximum of two `Sq` instances.

These operators and functions make the `Sq` type a powerful tool in the fixed-point math library, allowing for efficient and safe mathematical computations. Each operation is optimized to leverage the static nature of `Sq` values, ensuring computations are both fast and reliable, with checks and balances performed at compile-time to prevent runtime errors and ensure type safety.

Unary Operators

Unary Plus (+)

The unary plus operator in the context of the `Sq` type simply copies the `Sq` variable `v`. Typically, the unary plus operator is used for integral promotion, but this concept does not apply to the `Sq` type. The inclusion of this operator is mainly for the sake of completeness, ensuring that the `Sq` type has a consistent set of unary operators.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*Sq::realMin*
realMax	*Sq::realMax*
value	*v.value*

Example:

```
u32sq16<0., 10000.> sqVar = 5678.9_u32sq16;
auto sqVarCopy = +sqVar;  // Copies sqVar without any change in value and type
```

Unary Minus (-)

The unary minus operator inverts the sign of the value `v` and the limits of the `Sq` input type. This transformation essentially mirrors the value range around the origin.

Constraints:

For a signed base type, the minimum integer must not be within the value range because its negation will cause an overflow, resulting in the same negative value. This is why the default real value range is symmetric around 0. For example, an `i8q2<>` has a default symmetric real value range of -31.75 to 31.75. Although a type like `i8q2<-32., 31.75>` can still be manually declared, the negation operator is not available for this type.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*-Sq::realMax*
realMax	*-Sq::realMin*
value	*-v.value*

Example:

```
i16sq7<-100., 200.> sqVar = 150.0_i16sq7;
auto sqVarInverse = -sqVar;  // i16sq7<-200., 100.>, real value -150.
```

Binary Operators

Note: Although multiplication and division are used for rescaling *value* in the following sections for clarity, the actual implementation relies on the `s2s` function. By default, this function uses shift operations unless multiplication/division was explicitly selected. See Helpers for details.

Addition (+)

Adds two `Sq` instances `lhs` and `rhs`, producing a new `Sq` instance with a proper base type, the larger resolution, the limits added together, and the resultant value.

Output:

`Sq`	
base_t	*smallest integer fitting the resulting range, no smaller than any input; signed if either SqLhs or SqRhs is signed, otherwise unsigned*
f	*max(SqLhs::f, SqRhs::f)*
realMin	*SqLhs::realMin + SqRhs::realMin*
realMax	*SqLhs::realMax + SqRhs::realMax*
value	*lhs.value * 2^(f - SqLhs::f) + rhs.value * 2^(f - SqRhs::f)*

Example:

```
i16sq7<-100., 200.> sq1 = 150.0_i16sq7;
u16sq7<0., 100.> sq2 = 50.0_u16sq7;
auto sqAdded = sq1 + sq2;  // i16sq7<-100., 300.>, real value 200.
```

Subtraction (-)

Subtracts the right-hand side `Sq` instance `rhs` from the left-hand side `Sq` instance `lhs`, producing a new `Sq` object with a proper base type, the larger resolution, the limits subtracted, and the resultant value.

Output:

`Sq`	
base_t	*smallest integer fitting the resulting range, no smaller than any input; signed if either SqLhs or SqRhs is signed, otherwise unsigned*
f	*max(SqLhs::f, SqRhs::f)*
realMin	*min(SqLhs::realMin - SqRhs::realMax, SqRhs::realMin - SqLhs::realMax)*
realMax	*max(SqLhs::realMax - SqRhs::realMin, SqRhs::realMax - SqLhs::realMin)*
value	*lhs.value * 2^(f - SqLhs::f) - rhs.value * 2^(f - SqRhs::f)*

Example:

```
i16sq7<-100., 200.> sq1 = 150.0_i16sq7;
u16sq7<0., 100.> sq2 = 50.0_u16sq7;
auto sqSub = sq1 - sq2;  // i16sq7<-200., 200.>, real value 100.
```

Multiplication (*)

Sq * Sq

Multiplies two `Sq` instances `lhs` and `rhs`, producing a new `Sq` object with a proper base type, the larger resolution, the multiplied limits, and the resultant value.
During multiplication, an intermediate calculation type is used, which has twice the size and the sign of the unpromoted, common base type derived from the two input types. For instance, if the input base types are `int8_t` and `uint8_t`, the common base type and calculation type would be `uint8_t` and `uint16_t`, respectively. Even if one of the operands is negative and the calculation type is unsigned, the operation will still yield the correct result as integer wrapping ensures the desired result when cast to the signed base type of the resultant `Sq` type.

Formula:

$$(a * b)_{real} \iff \left((a * 2^f) * (b * 2^f) * 2^{-f} = (a * b) * 2^f\right)_{scaled}$$

Output:

`Sq`	*lhsMin = SqLhs::realMin, lhsMax = SqLhs::realMax, rhsMin = SqRhs::realMin, rhsMax = SqRhs::realMax*
base_t	*smallest integer fitting the resulting range, no smaller than any input; signed if either SqLhs or SqRhs is signed, otherwise unsigned*
f	*max(SqLhs::f, SqRhs::f)*
realMin	*min(min(lhsMin*rhsMax, rhsMin*lhsMax), min(lhsMin*rhsMin, lhsMax*rhsMax))*
realMax	*max(max(lhsMax*rhsMin, rhsMax*lhsMin), max(lhsMin*rhsMin, lhsMax*rhsMax))*
value	*(lhs.value * 2^(f - SqLhs::f) * rhs.value * 2^(f - SqRhs::f)) * 2^-f*

Example:

```
i16sq7<-200., 100.> sq1 = -150.0_i16sq7;
u16sq7<0., 100.> sq2 = 50.0_u16sq7;
auto sqMul = sq1 * sq2;  // i32sq7<-20000., 10000.>, real value -7500.
```

Sq * Integral Constant

Multiplies an `Sq` instance `v` with an integral constant. The result is a new `Sq` type with a proper base type, the same resolution, and both the limits and the value multiplied with the integral constant.

Note that plain integer literals do not work, as the C++ language does not support the necessary operator overloads with integers in the compile-time context (yet). The `_ic` literal provided by this library can be used to construct an integral constant with `unsigned int` type, which is less tedious than the `std::integral_constant<T,v>` trait from the standard library.

Output:

`Sq`	
base_t	*smallest integer fitting the resulting range, no smaller than any input; signed if either Sq or the integral constant is signed, otherwise unsigned*
f	*Sq::f*
realMin	*min(Sq::realMin * ic, Sq::realMax * ic)*
realMax	*max(Sq::realMin * ic, Sq::realMax * ic)*

`Sq`	
value	*v.value * ic*

Example:

```
i16sq7<-200., 100.> sq1 = -150.0_i16sq7;
u16sq7<0., 100.> sq2 = 50.0_u16sq7;
auto sqMul1 = sq1 * 3_ic;    // i32sq7<-600., 300.>, real value: -450.0
auto sqMul2 = -40_ic * sq2;  // i32sq7<-4000., 0.>, real value: -2000.0
```

Division (/)

Sq / Sq

Divides one `Sq` instance `lhs` by another, `rhs`, producing a new `Sq` object that properly balances the type, resolution, and limits based on the divisor and dividend. The division uses an intermediate calculation type that can handle a divisor in 2^{2f} representation, ensuring adequate size and sign handling.

Constraints:

The dividend (`SqRhs`) must not have values from the range $(-1, +1)$ in its real value range. This constraint is crucial as it prevents division by zero and avoids any undesired expansion of the resulting value range that could lead to significant restrictions on further computations.

Formula:

$$(a/b)_{real} \iff \left((a * 2^f) * 2^f / (b * 2^f) = (a/b) * 2^f\right)_{scaled}$$

Output:

`Sq`	*lhsMin = SqLhs::realMin, lhsMax = SqLhs::realMax, rhsMin = SqRhs::realMin, rhsMax = SqRhs::realMax*
base_t	*smallest integer fitting the resulting range, no smaller than any input; signed if either SqLhs or SqRhs is signed, otherwise unsigned*
f	*max(SqLhs::f, SqRhs::f)*

`Sq`	*lhsMin = SqLhs::realMin, lhsMax = SqLhs::realMax, rhsMin = SqRhs::realMin, rhsMax = SqRhs::realMax*
realMin	*min(min(lhsMin/rhsMax, lhsMin/rhsMin), min(lhsMax/rhsMin, lhsMax/rhsMax))*
realMax	*max(max(lhsMin/rhsMax, lhsMin/rhsMin), max(lhsMax/rhsMin, lhsMax/rhsMax))*
value	*lhs.value * 2^(2*f - SqLhs::f) / [rhs.value * 2^(f - SqRhs::f)]*

Example:

```
i16sq7<-200., 100.> sq1 = -150.0_i16sq7;
u16sq7<1.0, 100.> sq2 = 50.0_u16sq7;
auto sqDiv = sq1 / sq2;  // i16sq7<-200., 100.>, real value -3.0
```

Sq / Integral Constant

Divides an `Sq` instance `v` by an integral constant. The result maintains the `Sq` type's base type and resolution, adjusting limits and value based on the constant.

Note that plain integer literals do not work, as the C++ language does not support the necessary operator overloads with integers in the compile-time context (yet). The `_ic` literal provided by this library can be used to construct an integral constant with `unsigned int` type, which is less tedious than the `std::integral_constant<T,v>` trait from the standard library.

Output:

`Sq`	
base_t	*smallest integer fitting the resulting range, no smaller than any input; signed if either Sq or the integral constant is signed, otherwise unsigned*
f	*Sq::f*
realMin	*min(Sq::realMin / ic, Sq::realMax / ic)*
realMax	*max(Sq::realMin / ic, Sq::realMax / ic)*
value	*v.value / ic*

Example:

```
i16sq7<-200., 100.> sq1 = -150.0_i16sq7;
auto sqDiv = sq1 / 5_ic;  // i16sq7<-40., 20.>, real value: -30.0
```

Integral Constant / Sq

Divides an integral constant by an `Sq` instance `v`. The result maintains the `Sq` type's base type and resolution, adjusting limits and value based on the constant. The division uses an intermediate calculation type that can handle a divisor in 2^{2f} representation, ensuring adequate size and sign handling.

Constraints:

The dividend (`v`) must not have values from the range $(-1, 1)$ in its real value range. This constraint is crucial as it prevents division by zero and avoids any undesired expansion of the resulting value range that could lead to significant restrictions on further computations.

Output:

`Sq`	
base_t	*smallest integer fitting the resulting range, no smaller than any input; signed if either Sq or the integral constant is signed, otherwise unsigned*
f	*Sq::f*
realMin	*min(ic / Sq::realMin, ic / Sq::realMax)*
realMax	*max(ic / Sq::realMin, ic / Sq::realMax)*
value	*ic * 2^(2f) / v.value*

Example:

```
i16sq7<-200., -10.> sq = -150.0_i16sq7;
auto sqDiv = -1500_ic / sq;  // i16sq7<7.5, 150.>, real value: -10.0
```

Modulo (%)

Calculates the modulo of one `Sq` instance `lhs` by another, `rhs`, resulting in a new `Sq` object with adjusted type and limits based on the operands. This operation uses the unpromoted common type of the two input types as intermediate calculation type, corrected for the potential change in scaling, ensuring adequate size and sign handling.

Formula:

$$(a \bmod b)_{real} \iff \left((a * 2^f) \bmod (b * 2^f) = (a \bmod b) * 2^f\right)_{scaled}$$

Constraints:

The dividend (`SqRhs`) must not have any values from the range $(-resolution, resolution)$ in its real value range. This constraint is crucial as it prevents a modulo zero, as this is not defined.

Output:

`Sq`	*lhsMin = SqLhs::realMin, lhsMax = SqLhs::realMax, rhsMin = SqRhs::realMin, rhsMax = SqRhs::realMax*
base_t	*smallest integer fitting the resulting range, no smaller than any input; signed if either SqLhs or SqRhs is signed, otherwise unsigned*
f	*max(SqLhs::f, SqRhs::f)*
realMin	*max(lhsMin, signum(lhsMin) * max(abs(rhsMin), abs(rhsMax)))*
realMax	*min(lhsMax, signum(lhsMax) * max(abs(rhsMin), abs(rhsMax)))*
value	*(lhs.value * 2^(f - SqLhs::f)) % (rhs.value * 2^(f - SqRhs::f))*

Example:

```
i16sq7<-200., 100.> sq1 = -150.0_i16sq7;
u16sq7<10., 100.> sq2 = 45.0_u16sq7;
auto sqMod = sq1 % sq2;  // i16sq7<-100., 100.>, real value -15.0
```

Comparison Operators

Equality (==) and Inequality (!=)

Checks if two `Sq` instances `lhs` and `rhs` are equal or not, considering their scaled values.

Constraints:

- Both `Sq` instances must have the same resolution (`f`). If the resolutions are different, the operation will automatically convert them to the larger resolution before comparison.
- Both `Sq` instances must have the same signedness. If they have different signedness, the left-hand side (lhs) type must have a larger size compared to the right-hand side (rhs) type.

Output:

`bool`	
value	*true* if scaled values are equal, otherwise *false*

Example:

```
i32sq7<-100., 200.> sq1 = 150.0_i32sq7;
u16sq8<0., 100.> sq2 = 50.0_u16sq8;
bool isEqual = (sq1 == sq2);  // false
bool isNotEqual = (sq1 != sq2);  // true

u16sq7<0., 150.> sq3 = 150.0_u16sq7;
bool isEqual2 = (sq1 == sq3);  // true
```

Ordering (Spaceship Operator <=>)

Performs a three-way comparison of two `Sq` instances `lhs` and `rhs`, producing a result that indicates whether the first is less than, equal to, or greater than the second.

Constraints:

- Both `Sq` instances must have the same resolution (`f`). If the resolutions are different, the operation will automatically convert them to the larger resolution before comparison.
- Both `Sq` instances must have the same signedness. If they have different signedness, the left-hand side (lhs) type must have a larger size compared to the right-hand side (rhs) type.

Output:

`strong::ordering`	
value	depending on the values one of: *less*, *equal*, *greater*

Example:

```
i32sq7<-100., 200.> sq1 = 150.0_i32sq7;
u16sq7<0., 100.> sq2 = 50.0_u16sq7;
u16sq7<0., 150.> sq3 = 150.0_u16sq7;
i16sq7<-200., 100.> sq4 = -50.0_i16sq7;

auto cmpResult1 = (sq1 <=> sq2);  // std::strong_ordering::greater
auto cmpResult2 = (sq1 <=> sq3);  // std::strong_ordering::equal
auto cmpResult3 = (sq4 <=> sq1);  // std::strong_ordering::less
```

These operators allow for intuitive and straightforward comparison of `Sq` instances, ensuring that their underlying scaled values are considered while respecting the resolution and type constraints.

Ordering (<, >, <=, >=)

Compares two `Sq` instances `lhs` and `rhs`, determining if one is less than, greater than, less than or equal to, or greater than or equal to the other based on their scaled values.

Note: These operators are generated by the compiler from the spaceship operator implementation.

Constraints:

- Both `Sq` instances must have the same resolution (`f`). If the resolutions are different, the operation will automatically convert them to the larger resolution before comparison.
- Both `Sq` instances must have the same signedness. If they have different signedness, the left-hand side (lhs) type must have a larger size compared to the right-hand side (rhs) type.

Output:

`bool`	
value	*true* or *false* based on the comparison

Example:

```
i32sq10<-100., 200.> sq1 = 150.0_i32sq10;
u16sq7<0., 100.> sq2 = 50.0_u16sq7;
u16sq7<0., 150.> sq3 = 150.0_u16sq7;
i16sq7<-200., 100.> sq4 = -50.0_i16sq7;

bool isLessThan = (sq1 < sq2);  // false
bool isGreaterThan = (sq1 > sq2);  // true
bool isLessThanOrEqual = (sq1 <= sq3);  // true
bool isGreaterThanOrEqual = (sq4 >= sq1);  // false

// usage in conditional statements
if (sq1 < sq2) {
    // sq1 is less than sq2
}
else if (sq1 > sq2) {
    // sq1 is greater than sq2
}
else if (sq1 <= sq2) {
    // sq1 is less than or equal to sq2
}
else if (sq1 >= sq2) {
    // sq1 is greater than or equal to sq2
}
else {}
```

Shift Operators

The shift operators facilitate shifting `Sq` instances left or right by a specified number of bits, leveraging integral constants (using the `_ic` literal) for compile-time evaluation. This ensures that the resolution and type constraints are maintained, while the limits and values are adjusted accordingly.

Left Shift (<<)

Shifts the scaled value of an `Sq` instance `v` to the left by a specified number of bits, given as an integral constant (using the `_ic` literal). The result is a new `Sq` type with the same base type, adjusted limits, and the scaled value shifted. Effectively, a left shift is similar to a multiplication by a power of 2.

Constraints:

The shift amount must be a non-negative integral constant.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*(Sq::scaledMin << ic) * 2^{-f}*
realMax	*(Sq::scaledMax << ic) * 2^{-f}*
value	*v.value << ic*

Example:

```
i32sq16<-100., 200.> sq = 150.0_i32sq16;
auto sqShiftLeft = sq << 2_ic;  // i32sq16<-400., 800.>, real value 600.
```

Right Shift (>>)

Shifts the scaled value of an `Sq` instance `v` to the right by a specified number of bits, given as an integral constant (using the `_ic` literal). The result is a new `Sq` type with the same base type, adjusted limits, and the scaled value shifted. Effectively, a right shift is similar to a division by a power of 2, with the result rounded towards $-\infty$ to the nearest integer.

Constraints:

The shift amount must be a non-negative integral constant.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*(Sq::scaledMin >> ic) * 2^{-f}*
realMax	*(Sq::scaledMax >> ic) * 2^{-f}*
value	*v.value >> ic*

Example:

```
i16sq4<-200., 1000.> sq = -147.7_i16sq4;  // scaled: -2363
auto sqShiftRight = sq >> 3_ic;  // i16sq4<-25., 125.>, sc: -296, real: -18.5
```

Clamping Functions

Clamping functions are essential for maintaining values within specific ranges, ensuring efficient and safe mathematical computations with `Sq` instances. Each operation is optimized to leverage the static nature of `Sq` values, ensuring computations are both fast and reliable, with checks and balances performed at compile-time to prevent runtime errors and ensure type safety.

Note: Although multiplication and division are used for rescaling *value* in the following sections for clarity, the actual implementation relies on the `s2s` function. By default, this function uses shift operations unless multiplication/division was explicitly selected. See Helpers for details.

Clamp to Minimum

Ensures the `Sq` instance `v` does not fall below a specified minimum value. This function is available in two versions: one for runtime limits and one for compile-time limits.

Runtime Clamp to Minimum

Constraints:

- The minimum value (`lo`) of type `SqLo` must be implicitly convertible to the type of the `Sq` instance being clamped (`v`). **Implicitly Convertible** from A to B means:
  - A and B are `Sq` types with the same base type.
  - A to B is scalable: the number of shifted bits is at most the number of bits in each type.
  - The real value range of B includes the real value range of A.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*SqLo::realMin*

`Sq`	
realMax	*Sq::realMax*
value	*(v < lo) ? lo.value * 2^(f - SqLo::f) : v.value*

Example:

```
i16sq7<-100., 200.> sq = 50.0_i16sq7;
i16sq7<50., 150.> sqLo = 100.0_i16sq7;
auto clampedMin = clampLower(sq, sqLo);  // i16sq7<50., 200.>, real: 100.
```

Compile-Time Clamp to Minimum

Compile-time limits can be either real `double` values or `Sq` literals.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*realMin*
realMax	*Sq::realMax*
lo	*realMin * 2^f*
value	*(v.value < lo) ? lo : v.value*

Example:

```
i16sq2<-100., 200.> sq = -100.0_i16sq2;
auto clampedMin = clampLower<-50.0>(sq);  // i16sq2<-50., 200.>, real: -50.
auto clampedMin2 = clampLower<20.25_i16q2>(sq);  // i16sq2<20.25, 200.>, real: 20.25
```

Clamp to Maximum

Ensures the `Sq` instance `v` does not exceed a specified maximum value. This function is available in two versions: one for runtime limits and one for compile-time limits.

Runtime Clamp to Maximum

Constraints:

- The maximum value (`hi`) of type `SqHi` must be implicitly convertible to the type of the `Sq` instance being clamped (`v`). **Implicitly Convertible** from A to B means:
  - A and B are `Sq` types with the same base type.
  - A to B is scalable: the number of shifted bits is at most the number of bits in each type.
  - The real value range of B includes the real value range of A.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*Sq::realMin*
realMax	*SqHi::realMax*
value	*(hi < v) ? hi.value * 2^(f - SqHi::f) : v.value*

Example:

```
i16sq7<-100., 200.> sq = 180.0_i16sq7;
i16sq7<-100., 150.> sqHi = 140.0_i16sq7;
auto clampedMax = clampUpper(sq, sqHi);  // i16sq7<-100., 150.>, real: 140.
```

Compile-Time Clamp to Maximum

Compile-time limits can be either real `double` values or `Sq` literals.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*Sq::realMin*
realMax	*realMax*
hi	*realMax * 2^f*
value	*(hi < v.value) ? hi : v.value*

Example:

```
i16sq1<-100., 200.> sq = 180.5_i16sq1;
auto clampedMax = clampUpper<160.0>(sq);  // i16sq7<-100., 160.>, real: 160.
auto clampedMax2 = clampUpper<10.5_i16sq1>(sq);  // i16sq7<-100., 10.5>, real: 10.5
```

Clamp to Range

Restricts the `Sq` instance `v` within a specified minimum and maximum range, combining both the clamp to minimum and clamp to maximum functionalities. This function is available in two versions: one for runtime limits and one for compile-time limits.

Runtime Clamp to Range

Constraints:

The minimum value (`lo`) of type `SqLo` and maximum value (`hi`) of type `SqHi` must be implicitly convertible to the type of the `Sq` instance being clamped (`v`). **Implicitly Convertible** from A to B means:

- A and B are `Sq` types with the same base type.
- A to B is scalable: the number of shifted bits is at most the number of bits in each type.
- The real value range of B includes the real value range of A.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*SqLo::realMin*
realMax	*SqHi::realMax*
value	*(v < lo) ? lo.value * 2^(f - SqLo::f)* *: (hi < v) ? hi.value * 2^(f - SqHi::f) : v.value*

Example:

```
i16sq7<-100., 200.> sq = -20.0_i16sq7;
i16sq7<-50., 200.> sqLo = 10.0_i16sq7;
i16sq7<-100., 180.> sqHi = 150.0_i16sq7;
auto clamped = clamp(sq, sqLo, sqHi);  // i16sq7<-50., 180.>, real: 10.
```

Compile-Time Clamp to Range

Compile-time limits can be either real `double` values or `Sq` literals.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*realMin*
realMax	*realMax*
lo	*realMin * 2^f*
hi	*realMax * 2^f*
value	*(v.value < lo) ? lo : (hi < v.value) ? hi : v.value*

Example:

```
i16sq7<-100., 200.> sq = -88.5_i16sq7;
auto clamped = clamp<-40., 200.>(sq);  // i16sq7<-40., 200.>, real: -40.
auto clamped2 = clamp<-20_i16sq7, 150_i16sq7>(sq);  // i16sq7<-20., 150.>, real: -20.
```

Mathematical Functions

Note: Although multiplication and division are used for rescaling *value* in the following sections for clarity, the actual implementation relies on the `s2s` function. By default, this function uses shift operations unless multiplication/division was explicitly selected. See Helpers for details.

Absolute (abs)

Returns the absolute value of an `Sq` instance `v`. Produces a new `Sq` instance with a proper unsigned base type and absolute limits, and the resultant absolute value.

Constraints:

If the type is signed, the corresponding minimum integral value must not be in the value range. For instance, if `int8_t` is the base type, a scaled value of -128 is not allowed. This is because the absolute value of the minimum integer exceeds the value range of the signed type.

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*0.0*, if 0 is in real range of source type, otherwise *min(abs(Sq::realMin), abs(Sq::realMax))*
realMax	*max(abs(Sq::realMin), abs(Sq::realMax))*
value	*| v.value |*

Example:

```
i16sq7<-100., 200.> sq = -75.0_i16sq7;
auto absValue = abs(sq);  // u16sq7<0., 200.>, real value 75.0
```

Square (sqr)

Computes the square (x^2) of an `Sq` instance `v`. Produces a new `Sq` instance with a proper base type and squared limits, and the resultant squared value.
During computation, an intermediate calculation type is used, which has twice the size and the sign of the source base type.

Formula:

$$x^2{}_{real} \Longleftrightarrow \left((x * 2^f) * (x * 2^f) * 2^{-f} = x^2 * 2^f\right)_{scaled}$$

Output:

`Sq`	
base_t	*smallest integer fitting the resulting range, no smaller than int32_t;* *signed if the common type of int32_t and the source base type is signed, otherwise uns*
f	*Sq::f*
realMin	*0.0*, if 0 is in real range of source type, otherwise *min(Sq::realMin*Sq::realMin, Sq::realMax*Sq::realMax)*
realMax	*max(Sq::realMin*Sq::realMin, Sq::realMax*Sq::realMax)*
value	*v.value * v.value / 2^f*

Example:

```
i16sq7<-100., 200.> sq = -10.0_i16sq7;
auto squaredValue = sqr(sq);  // i32sq7<0., 40000.>, real value 100.0
```

Square Root (sqrt)

Calculates the square root of an `Sq` instance `v`. The result is an instance of a new `Sq` type, with an appropriate base type, the roots of the source limits, and the resultant value root. The computation uses `uint64_t` as an intermediate calculation type, and the square root of the value is computed using a binary search algorithm `isqrt` taken from *Hacker's Delight, 2nd ed.*

The root of each limit is approximated at compile-time using the inverse of the famous inverse square root implementation from *Quake III Arena*, optimized for `uint64_t` and utilizing the Newton-Raphson method. The new lower limit is then rounded down and the new upper limit is rounded up to correct for (most) approximation inaccuracies and to ensure clean limits in the resulting type.

Constraints:

- The size of the source base type must be smaller than or equal to the size of `uint32_t`.
- The scaling `f` must be smaller than the number of digits in the source base type.
- `Sq::realMin` must be greater than or equal to 0, i.e. there must not be any negative values in the real value range.

Formula:

$$sqrt(x)_{real} \iff \left(\left((x * 2^f) * 2^f \right)^{1/2} = x^{1/2} * 2^f \right)_{scaled}$$

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*floor(sqrt(Sq::realMin))*
realMax	*ceil(sqrt(Sq::realMax))*
value	*isqrt(v.value * 2^f)*

Example:

```
i16sq7<0., 200.> sq = 100.0_i16sq7;
auto sqrtValue = sqrt(sq);  // i16sq7<0., 15.>, real value 10.0
```

Inverse Square Root (rsqrt)

Calculates the reciprocal of the square root of an `Sq` instance `v`, commonly used in graphics and physics calculations to improve performance. This results in a new `Sq` instance with an appropriate base type, the reverse square root of the limits, and the resultant value.

The runtime implementation uses the `sqrt` function for the square root, which employs `uint64_t` as an intermediate calculation type and computes the root using the `isqrt` binary search algorithm from *Hacker's Delight, 2nd ed.* The inverse root of each limit is approximated at compile-time using the famous inverse square root implementation from *Quake III Arena*, optimized for `uint64_t` and utilizing the Newton-Raphson method. The new lower limit is then rounded down and the new upper limit is rounded up to correct for (most) approximation inaccuracies and to ensure clean limits in the resulting type.

Note that the result will saturate at the maximum representable real value $thMax$ if the runtime scaled value $x * 2^f$ is smaller than or equal to $limit = 2^f/thMax^2$.

Constraints:

- The size of the source base type must be smaller than or equal to the size of `uint32_t`.
- The scaling `f` must be smaller than the number of digits in the source base type.
- To avoid division by 0, `Sq::realMin` must be equal to or greater than the type's resolution, i.e. only scaled values >= 1 are permitted.

Formula:

$$sqrt^{-1}(x)_{real} \Longleftrightarrow \left(2^{2f}/\left((x * 2^f) * 2^f\right)^{1/2} = 2^f/x^{1/2}\right)_{scaled}$$

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*floor(rsqrt(Sq::realMax))*
realMax	*min(thMax, ceil(rsqrt(Sq::realMin)))* ; *thMax* is the largest possible real value

`Sq`	
value	*thMax * 2^f* , if *v.value < limit*, else *2^2f / sqrt(v)*

Example:

```
i16sq7<1., 100.> sq = 25.0_i16sq7;
auto rsqrtValue = rsqrt(sq);  // i16sq7<0.0, 1.0>, real value 0.195313

auto sq2 = i8sq6<i8sq6<>::resolution, 1.>::fromScaled< 1 >();
auto rsqrtValue2 = rsqrt(sq2);  // i8sq6<0., 1.984375>, real value 1.984375
```

Cube (cube)

Computes the cube (x^3) of an `Sq` instance `v`. Produces a new `Sq` instance with a proper base type and cubedc limits, and the resultant cubed value.
During computation, an intermediate calculation type is used, which has twice the size and the sign of the source base type.

Formula:

$$x^3{}_{real} \Longleftrightarrow \left((x * 2^f) * (x * 2^f) * 2^{-f} * (x * 2^f) * 2^{-f} = x^3 * 2^f\right)_{scaled}$$

Output:

`Sq`	*l1=Sq::realMin*Sq::realMin*Sq::realMin, l2=Sq::realMin*Sq::realMin*Sq::realMax, l3=Sq::realMin*Sq::realMax*Sq::realMax, l4=Sq::realMax*Sq::realMax*Sq::realMax*
base_t	*smallest integer fitting the resulting range, no smaller than int32_t; signed if the common type of int32_t and the source base type is signed, otherwise uns*
f	*Sq::f*
realMin	*min(l1, l2, l3, l4)*
realMax	*max(l1, l2, l3, l4)*
value	*sqr(v) * v.value / 2^f*

Example:

```
i16sq7<-100., 100.> sq = -5.0_i16sq7;
auto cubeValue = cube(sq);  // i32sq7<-1e6, 1e6>, real value -125.0
```

Cube Root (cbrt)

Calculates the cube root of an `Sq` instance `v`. The result is an instance of a new `Sq` type, with the same base type, the roots of the source limits, and the resultant value root. The computation uses `uint64_t` as an intermediate calculation type, and the cube root of the value is computed using an algorithm based on `icbrt` from *Hacker's Delight, 2nd ed.*

The compile-time cube root of each limit is approximated using a binary search algorithm, with a target accuracy of `1e-6` but at most 200 iterations. The new lower limit is then rounded down and the new upper limit is rounded up to correct for (most) approximation inaccuracies and to ensure clean limits in the resulting type.

Constraints:

- The size of the source base type must be smaller than or equal to the size of `uint32_t`.
- The scaling `f` must be smaller than or equal to 16.
- `Sq::realMin` must be greater than or equal to 0, i.e. there must not be any negative values in the real value range.

Formula:

$$cbrt(x)_{real} \Longleftrightarrow \left(\left((x * 2^f) * 2^f * 2^f \right)^{1/3} = x^{1/3} * 2^f \right)_{scaled}$$

Output:

`Sq`	
base_t	*Sq::base_t*
f	*Sq::f*
realMin	*floor(cbrt(Sq::realMin))*
realMax	*ceil(cbrt(Sq::realMax))*

`Sq`	
value	*icbrt(v.value * 2^f * 2^f)*

Example:

```
i16sq7<0., 200.> sq = 125.0_i16sq7;
auto cbrtValue = cbrt(sq);  // i16sq7<0., 6.>, real value 5.0
```

Minimum (min)

Determines the minimum of two similar `Sq` instances `v1` and `v2`. Produces a new `Sq` instance with the same base type and scaling, the minimum limits and the resultant minimum value.

Constraints:

Both `Sq` instances must be similar, i.e. have the same base type and scaling.

Output:

`Sq`	
base_t	*Sq1::base_t*
f	*Sq1::f*
realMin	*min(Sq1::realMin, Sq2::realMin)*
realMax	*min(Sq1::realMax, Sq2::realMax)*
value	*(v1.value > v2.value) ? v2.value : v1.value*

Example:

```
i16sq7<-100., 200.> sq1 = 50.0_i16sq7;
i16sq7<-250., 250.> sq2 = 150.0_i16sq7;
auto minValue = min(sq1, sq2);  // i16sq7<-250., 200.>, real value 50.0
```

Maximum (max)

Determines the maximum of two similar `Sq` instances `v1` and `v2`. Produces a new `Sq` instance with the same base type and scaling, the maximum limits and the resultant maximum value.

Constraints:

Both `Sq` instances must be similar, i.e. have the same base type and scaling.

Output:

`Sq`	
base_t	*Sq1::base_t*
f	*Sq1::f*
realMin	*max(Sq1::realMin, Sq2::realMin)*
realMax	*max(Sq1::realMax, Sq2::realMax)*
value	*(v1.value < v2.value) ? v2.value : v1.value*

Example:

```
i16sq7<-100., 200.> sq1 = 50.0_i16sq7;
i16sq7<-200., 250.> sq2 = 150.0_i16sq7;
auto maxValue = max(sq1, sq2);  // i16sq7<-100., 250.>, real value 150.0
```

Practical Example

```cpp
#include <iostream>

#include <fpm.hpp>

using namespace fpm::types;
using Ovf = fpm::Ovf;


using pos_t = i32q16<-2000., 2000. /* mm */>;
FPM_Q_BIND_LITERAL(pos_t, mm);

using speed_t = i32q16<-300., 300. /* mm/s */>;
FPM_Q_BIND_LITERAL(speed_t, mm_p_s);

using accel_t = i32q16<-200., 200. /* mm/s2 */>;
FPM_Q_BIND_LITERAL(accel_t, mm_p_s2);

using mtime_t = u32q20<0., 2000. /* s */, Ovf::allowed>;
FPM_Q_BIND_LITERAL(mtime_t, s);
using ts_t = mtime_t::clamp_t<0., .01>;  // u32q20


void accel(pos_t &s, speed_t &v, accel_t const a, mtime_t const time, ts_t const dt) {
    for (mtime_t t = 0_s; t < time; t = t + dt) {
        auto dv = a * dt;  // i32sq20<-2., 2.>
        auto ds = v * dt;  // i32sq20<-3., 3.>

        // note: computation results are in Sq representation;
        // explicit conversion back to Q representation is necessary
        v = speed_t::fromSq<Ovf::clamp>(v + dv);
        s = pos_t::fromSq<Ovf::clamp>(s + ds);
    }
}


int main() {
    pos_t position = -10.2_mm;
    speed_t velocity = 0_mm_p_s;

    // acceleration of 100 mm/s^2 for 1 second
    accel(position, velocity, 100_mm_p_s2, 1_s, 1e-3_s);

    // Position: 39.7874, Velocity: 100.045
    std::cout << "Position: " << position.real() << "\n"
              << "Velocity: " << velocity.real() << std::endl;

    // copy position and velocity
    pos_t position2 = position;
    speed_t velocity2 = velocity;

    // clamp velocity2 to 80 mm/s
    velocity2 = clampUpper<80_mm_p_s>(velocity2);

    // deceleration of 50 mm/s^2 for 3 seconds, fewer steps
    accel(position2, velocity2, -50_mm_p_s2, 3_s, 1e-2_s);

    // Position2: 54.8479, Velocity2: -70.4908
    std::cout << "Position2: " << position2.real() << "\n"
```

```
              << "Velocity2: " << velocity2.real() << std::endl;

    // take minimum
    auto minPos = min(position, position2);
    auto minVel = min(velocity, velocity2);

    // minPos: 39.7874, minVel: -70.4908
    std::cout << "minPos: " << minPos.real() << "\n"
              << "minVel: " << minVel.real() << std::endl;

    return 0;
}
```