Document number: P3605R1
Project: Programming Language C++
Audience: SG6 (Numerics Study Group), LEWG, LWG

Nikita Sakharin, PJSC Sberbank <nikitasa1997@gmail.com>

Date: 2026-03-02

isqrt: A function to calculate integer square root of nonnegative integers

Contents

0. Revision history

Changes since R0

In accordance with the **SG6** feedback, the following changes were made:

- Add **C Compatibility** and **Constraints vs Mandates** sections to the **Design Considerations**;
- Make `noexcept` conditional;
- Replace *Mandates* with *Constraints*;
- Exclude `char` type from those that the function template can be instantiated for;
- Remove *Complexity*;
- Simplify *Returns*.

1. Abstract

This paper proposes to add an `isqrt` function (template) to calculate the integer square root of a nonnegative integer. Mathematically defined as:

$$\mathrm{isqrt}\left(n\right) = \left\lfloor \sqrt{n} \right\rfloor = \max\left\{k \in N : k^2 \leq n\right\}, \text{for } n \in N, \text{where } N = \left\{0, 1, 2, 3, \ \ldots\right\}.$$

`isqrt` of a nonnegative integer `n` is the greatest integer whose square is less than or equal to `n`.

2. Motivation

We will use the following notation:

- **Standard** is the N5032 Working Draft;
- `uintmax_t` is the type `std::uintmax_t` from header `<cstdint>`;
- `sqrt` is the function `std::sqrt` from header `<cmath>`; and
- `double` type is assumed to be the `binary64` type defined in the **ISO/IEC 60559:2020** (**IEEE 754-2019**) standard.

2.1. A common number-theoretic algorithm

The integer square root[1] is a useful number-theoretic primitive. For example, it is commonly applied in:

- **Primality test** and **Integer factorization** algorithms, such as **Trial division** and **Fermat's method**[2];
- **Cryptography** algorithms, such as block entanglement (non-linear transformation)[3];
- **Sqrt-decomposition** method[2]; and
- **Block Merge Sort** algorithm[4].

This algorithm is also pedagogically important. For example, it is published as the "Sqrt(x)"[5] problem on **LeetCode**.

2.2. Expressions `uintmax_t(sqrt(n))` and `isqrt(n)` give different results

There are numerous popular questions about integer square root in **C++** on **StackOverflow**. For example:

- "Fastest way to get the integer part of sqrt(n)?"[6]
- "Looking for an efficient integer square root algorithm for ARM Thumb2"[7]
- "How can you easily calculate the square root of an unsigned long long in C?"[8]
- "Determining if square root is an integer"[9]

Answers to the above questions often recommend a naive solution such as `uintmax_t(sqrt(n))` (or equivalent). At first glance, this may seem correct, because for "small" numbers, that expression actually gives the same result as the `isqrt(n)` function defined above. But for "large" numbers, these two expressions could give different results, even when the value of `n` is exactly representable in the floating-point type used for `sqrt` calculation. Here we define a number to be "large" when it is greater than $2^{\texttt{digits}}$, where `digits` is the number of mantissa bits of the type. Therefore, how "large" the number must be to cause a different result depends on the type used for calculation.

Consider, for example, the `double` type. Per section **§29.7.1 [cmath.syn]** of the **Standard**, this type is used for calculation whenever the argument of `sqrt` has an integer type. For the `double` type, the number of mantissa bits is `52`, so the value of `n` from the example is greater than 2^{52}. Let us take the number:

$$n = 67108865^2 - 1 = 4503599761588224 = 2^{52} + 2^{27},$$

which is exactly representable as a `double`. The square root of this number is:

$$\sqrt{n} = \sqrt{4503599761588224} = 67108864.9999999925494195...$$

According to the definition given above, the `isqrt(n)` call must discard the fractional part of the square root:

$$\text{isqrt}\left(n\right) = \left\lfloor\sqrt{n}\right\rfloor = \left\lfloor\sqrt{4503599761588224}\right\rfloor = \left\lfloor 67108864.9999999925494195...\right\rfloor = 67108864.$$

As an irrational number, $\sqrt{n}$ cannot be represented exactly as a `double`. Therefore, the call `sqrt(n)` returns the nearest (to the "correct") `double`-representable value. The two adjacent values of `double` type between which $\sqrt{n}$ is enclosed are:

$$67108865 - 2^{-26} = 67108864.999999985098838806152343750 < \sqrt{n} < 67108865.$$

These two values are shown in the table below, along with their representation in the `double` type. The rightmost column shows the absolute error of the `double` approximations to $\sqrt{n}$:

Value	Representation in `double` type			$\lvert\text{value} - \sqrt{n}\rvert$
	sign	exponent	mantissa	
67108864.999999985098838806152343750	0	10000011001	0000000000000000000000000011111111111111111111111111	**7.450580**7079461285×10^{-9}
67108865.0	0	10000011001	0000000000000000000000000100000000000000000000000000	**7.450580**4859015277×10^{-9}

Since the absolute error is smaller for the value `67108865.0`, this number is closer to the exact value of the square root, so:

$$\text{sqrt}\left(n\right) = \text{sqrt}\left(4503599761588224\right) = 67108865.$$

Therefore:

$$\text{uintmax\_t}\left(\text{sqrt}\left(n\right)\right) = \text{uintmax\_t}\left(\text{sqrt}\left(4503599761588224\right)\right) = \text{uintmax\_t}\left(67108865\right) = 67108865.$$

Thus:

$$\exists n = i^2 - 1 \text{ for some } i \in N\text{: uintmax\_t}\left(\text{sqrt}\left(n\right)\right) \neq \text{isqrt}\left(n\right).$$

The following table shows the least four values of the integer `n` for which the expressions `uintmax_t(sqrt(n))` and `isqrt(n)` give different results:

i	n = i² - 1	uintmax_t(sqrt(n)) = i	isqrt(n) = i - 1
67108865	4503599761588224	67108865	67108864
67108866	4503599895805955	67108866	67108865
67108867	4503600030023688	67108867	67108866
67108868	4503600164241423	67108868	67108867

Finally, if the `long double` type were implemented as an 80-bit floating-point type, then using it would solve the problem for 64-bit integers. However, if the `int128_t` type were added to the **Standard**, then the problem would arise again even for such an 80-bit `long double`. Therefore, the problem cannot be solved simply by using a wider floating-point type.

2.3. Prior Art

Several programming languages have a function (or class method) for calculating the integer square root:

- In **Java**, the `BigInteger` class has the `sqrt()` method[10];
- In **Python**, the `math` module has the `isqrt()` function[11];
- In **Ruby**, the `Integer` class has the `sqrt()` method[12]; and
- In **Rust**, primitive integer types have the `isqrt()` method[13].

3. Design Considerations

3.1 A new overload of `sqrt` cannot be added

Section **§29.7.1 [cmath.syn]** of the **Standard** defines overloads of the `sqrt` for an argument of integer type. Therefore, an overload of the `sqrt` function with an argument and a return value of integer type cannot be added.

3.2 The new function should be named `isqrt`

The **ISO/IEC 10967-2:2001** standard defines an integer square root function named `isqrt`, and we follow this standard's guidance.

3.3 `isqrt` function template can be instantiated for both `signed` and `unsigned` integer types

Mathematically, the integer square root function is defined for only non-negative integers. However, if the function template could not be instantiated for `signed` integers, it would be necessary to cast the argument to an `unsigned` type, even where it is known (e.g., by construction) that the `signed` integer is non-negative. Therefore, the function template should be able to be instantiated for both signed and unsigned integer types (including `signed char` and `unsigned char`, but not for `char`).

3.4 C Compatibility

Provided this proposal is merged to **C++**, the same function will be proposed to **C**. Therefore, a potential compatibility issue should be avoided.

First, the **C** programming language does not have function overloading. Instead, it uses naming conventions (prefixes/suffixes) to indicate the data types of arguments and return values of functions within families. Some of the conventions:

- `<math.h>`:
  - No suffix for `double`;
  - "**f**" suffix for `float`;
  - "**l**" suffix for `long double`.
- `<complex.h>`:
  - "**c**" prefix and no suffix for `double complex`;
  - "**c**" prefix and "**f**" suffix for `float complex`;
  - "**c**" prefix and "**l**" suffix for `long double complex`.
- `<stdlib.h>` and `<inttypes.h>`:
  - No prefix for `int`;
  - "**u**" prefix for `unsigned`;
  - "**l**" prefix for `long`;
  - "**ul**" prefix for `unsigned long`;
  - "**ll**" prefix for `long long`;
  - "**ull**" prefix for `unsigned long long`;
  - "**imax**" prefix for `intmax_t`;
  - "**umax**" prefix for `uintmax_t`;
- `<stdbit.h>` from **C23**:
  - "**_uc**" suffix for `unsigned char`;
  - "**_us**" suffix for `unsigned short`;
  - "**_ui**" suffix for `unsigned`;
  - "**_ul**" suffix for `unsigned long`;
  - "**_ull**" suffix for `unsigned long long`.

Thus, a decision to place the `isqrt` function into one of the **C** headers will force **WG14** to use specific prefixes/suffixes for this function, and possibly even the data types for which it is defined.

Second, the bit manipulation `<bit>` header was "backported" from **C++** to **C** under the name `<stdbit.h>`. And the **C++** function `popcount` was named `stdc_count_ones` in **C**, so not only was the `stdc_` prefix prepended, but the name itself was also changed. Also, the function `stdc_count_zeros`, added to **C** for consistency, has no "ancestor" at all. Thus, this shows that "backporting" a feature can change both its name and its headers.

In conclusion, the only guaranteed way to evade a potential compatibility issue is to avoid using existing **C** headers, such as `<cmath>` corresponding to `<math.h>` and others.

3.5 The header to which the function should be added

3.5.1 Header `<algorithm>`

Some of the functions contained in the `<algorithm>` header (such as `clamp`, `max`, `min`, and `minmax`) assume comparison operations are defined for their arguments. However, this header does not contain any function that expects its arguments to have arithmetic operations (except for iterator "arithmetic") such as addition, subtraction, multiplication, division and modulo. Since the `isqrt` function is defined in terms of arithmetic operations, this header is inappropriate.

3.5.2 Header `<cmath>`

The `<cmath>` header provides the standard mathematical function `sqrt`. At first glance, it seems reasonable to provide the `isqrt` function in the same header. However, the `<cmath>` header is strictly oriented towards functions with floating-point arguments, rather than functions with integer arguments.

Even more severe reason against it is the potential **C** compatibility issue, as described above, so this header is not suitable.

3.5.3 Header `<cstdlib>`

The `<cstdlib>` header is inappropriate due to the same reason as `<cmath>`.

3.5.4 Header `<numeric>`

The `<numeric>` header already contains "numerical" functions such as `gcd`, `lcm`, and `midpoint`, which are defined using arithmetic operations. The `isqrt` function is "numerical" by its nature and is also defined in terms of arithmetic. Additionally, selecting this header avoids potential compatibility issue, as it allows **WG14** to select possibly a different header and naming convention.

Following the **SG6** feedback, this header seems the most suitable one and it should be used for the `isqrt` function.

3.6 Constraints vs Mandates

A very careful choice should be made between *Constraints* and *Mandates*.

On the one hand, *Mandates* strictly limits the types `isqrt` function could be instantiated for. It prevents accidental misuse and provides a well-written, human-readable error message in case of a violation. The `<numeric>` header has functions `gcd` and `lcm` that use *Mandates*. But one reason why `gcd` and `lcm` use *Mandates* (rather than *Constraints*) could be that these functions had been added to the **Standard** in 2016 (N4606), while *Mandates*/*Constraints* were added only in 2018 (N4762).

On the other hand, *Constraints* is **SFINAE**-friendly, so it gives flexibility to use `isqrt` function with the `concept` in the `requires`-clause. Therefore, it provide room for future overloads (if needed). Several functions in the `<numeric>` header follow this logic: `midpoint`, and the recently added to the **Standard** saturation arithmetic functions — `add_sat`, `sub_sat`, `mul_sat`, `div_sat`, and `saturate_cast`.

Finally, the **SG6** advised to use *Constraints* in the feedback. All the arguments counted, the *Constraints* is the most appropriate choice.

4. Implementation Experience

Heron's method (a special case of **Newton's method**) for integers is discussed, for example, in the book "Hacker's Delight"[1]. For the initial estimate, the value $2^{\left\lceil \frac{\log_2(n)}{2} \right\rceil}$ is used, which is the least integer power of two that is greater than or equal to $\sqrt{n}$. Reference implementation:

```
using std::bit_width, std::is_unsigned_v, std::make_unsigned_t;

template<class T>
constexpr T isqrt(const T n) noexcept(is_unsigned_v<T>) {
    if (n <= T{1})
        return n;

    const auto exponent{(bit_width(make_unsigned_t<T>(n - 1)) + 1) >> 1};
    T i_current{0}, i_next(T{1} << exponent);
    do {
        i_current = i_next;
        i_next = T((i_current + n / i_current) >> 1);
    } while (i_next < i_current);

    return i_current;
}
```

5. Proposed Wording

Based on N5032:

5.1 Header `<version>` synopsis

Add to section **§17.3.2 Header `<version>` synopsis [version.syn]** the following:

```
#define __cpp_lib_is_within_lifetime 202306L // freestanding, also in <type_traits>
#define __cpp_lib_isqrt yyyymmL // freestanding, also in <numeric>
#define __cpp_lib_jthread 201911L // also in <stop_token>, <thread>
```

5.2 Header `<numeric>` synopsis

Add to section **§26.9 Header `<numeric>` synopsis [numeric.ops.overview]** the following:

```
// 26.10.17, saturation arithmetic
template<class T>
  constexpr T add_sat(T x, T y) noexcept;
template<class T>
  constexpr T sub_sat(T x, T y) noexcept;
template<class T>
  constexpr T mul_sat(T x, T y) noexcept;
template<class T>
  constexpr T div_sat(T x, T y) noexcept;
template<class T, class U>
  constexpr T saturate_cast(U x) noexcept;

// 26.10.18, integer square root
template<class T>
  constexpr T isqrt(T n) noexcept(is_unsigned_v<T>);
```

5.3 Integer square root

Add section **§26.10.18 Integer square root [numeric.ops.isqrt]** consisting of the following:

26.10.17.2 Casting **[numeric.sat.cast]**

```
template<class R, class T>
  constexpr R saturate_cast(T x) noexcept;
```

1 *Constraints*: `R` and `T` are signed or unsigned integer types (6.9.2).

2 *Returns*: If `x` is representable as a value of type `R`, `x`; otherwise, either the largest or smallest representable value of type `R`, whichever is closer to the value of `x`.

26.10.18 Integer square root **[numeric.ops.isqrt]**

```
template<class T>
  constexpr T isqrt(T n) noexcept(is_unsigned_v<T>);
```

1 *Constraints*: `T` is a signed or unsigned integer type.

2 *Preconditions*: `n` is non-negative.

3 *Returns*: $\lfloor\sqrt{n}\rfloor$.

6. Acknowledgements

Many thanks to Antony Polukhin for assistance in preparation of this paper.

Sincere thanks to Walter E. Brown for being so kind to give valuable advice and corrections to this proposal.

We are very grateful to Jan Schultke for his valuable suggestions and constructive feedback, which greatly improved the quality of this paper.

7. References

Hacker's Delight:

1. Henry S. Warren. 2012. Hacker's Delight (2nd. ed.). Addison-Wesley Professional.

Algorithm applications:

2. Antti Laaksonen. 2018. Guide to Competitive Programming: Learning and Improving Algorithms Through Contests (1st. ed.). Springer Publishing Company, Incorporated.
3. Boris S Verkhovsky. 2014. Integer Algorithms in Cryptology and Information Assurance. WORLD SCIENTIFIC.
4. Pok-Son Kim and Arne Kutzner. 2008. Ratio based stable in-place merging. In Proceedings of the 5th international conference on Theory and applications of models of computation (TAMC'08). Springer-Verlag, Berlin, Heidelberg, 246–257.

LeetCode:

5. Sqrt(x)

StackOverflow:

6. Fastest way to get the integer part of sqrt(n)?
7. Looking for an efficient integer square root algorithm for ARM Thumb2
8. How can you easily calculate the square root of an unsigned long long in C?
9. Determining if square root is an integer

Prior Art:

10. BigInteger.sqrt
11. math.isqrt
12. Integer.sqrt
13. i32.isqrt