

primesieve

7.6

Generated by Doxygen 1.9.1



<b>1 Main Page</b>	<b>1</b>
1.1 About	1
1.2 C++ API	1
1.3 C API	1
<b>2 Namespace Index</b>	<b>3</b>
2.1 Namespace List	3
<b>3 Hierarchical Index</b>	<b>5</b>
3.1 Class Hierarchy	5
<b>4 Class Index</b>	<b>7</b>
4.1 Class List	7
<b>5 File Index</b>	<b>9</b>
5.1 File List	9
<b>6 Namespace Documentation</b>	<b>11</b>
6.1 primesieve Namespace Reference	11
6.1.1 Detailed Description	12
6.1.2 Function Documentation	12
6.1.2.1 count_primes()	12
6.1.2.2 count_quadruplets()	13
6.1.2.3 count_quintuplets()	13
6.1.2.4 count_sextuplets()	13
6.1.2.5 count_triplets()	13
6.1.2.6 count_twins()	13
6.1.2.7 get_max_stop()	14
6.1.2.8 nth_prime()	14
6.1.2.9 set_num_threads()	14
6.1.2.10 set_sieve_size()	15
<b>7 Class Documentation</b>	<b>17</b>
7.1 primesieve::iterator Class Reference	17
7.1.1 Detailed Description	17
7.1.2 Constructor & Destructor Documentation	18
7.1.2.1 iterator()	18
7.1.3 Member Function Documentation	18
7.1.3.1 next_prime()	18
7.1.3.2 prev_prime()	18
7.1.3.3 skipto()	19
7.2 primesieve::primesieve_error Class Reference	20
7.2.1 Detailed Description	21
7.3 primesieve_iterator Struct Reference	21

7.3.1 Detailed Description . . . . .	21
<b>8 File Documentation</b>	<b>23</b>
8.1 iterator.h File Reference . . . . .	23
8.1.1 Detailed Description . . . . .	24
8.1.2 Function Documentation . . . . .	24
8.1.2.1 primesieve_next_prime() . . . . .	24
8.1.2.2 primesieve_prev_prime() . . . . .	25
8.1.2.3 primesieve_skipto() . . . . .	25
8.2 iterator.hpp File Reference . . . . .	25
8.2.1 Detailed Description . . . . .	27
8.3 primesieve.h File Reference . . . . .	27
8.3.1 Detailed Description . . . . .	29
8.3.2 Enumeration Type Documentation . . . . .	29
8.3.2.1 anonymous enum . . . . .	29
8.3.3 Function Documentation . . . . .	29
8.3.3.1 primesieve_count_primes() . . . . .	29
8.3.3.2 primesieve_count_quadruplets() . . . . .	30
8.3.3.3 primesieve_count_quintuplets() . . . . .	30
8.3.3.4 primesieve_count_sextuplets() . . . . .	30
8.3.3.5 primesieve_count_triplets() . . . . .	31
8.3.3.6 primesieve_count_twins() . . . . .	31
8.3.3.7 primesieve_generate_n_primes() . . . . .	31
8.3.3.8 primesieve_generate_primes() . . . . .	31
8.3.3.9 primesieve_get_max_stop() . . . . .	32
8.3.3.10 primesieve_nth_prime() . . . . .	32
8.3.3.11 primesieve_set_num_threads() . . . . .	33
8.3.3.12 primesieve_set_sieve_size() . . . . .	33
8.4 primesieve.hpp File Reference . . . . .	33
8.4.1 Detailed Description . . . . .	35
8.5 primesieve_error.hpp File Reference . . . . .	35
8.5.1 Detailed Description . . . . .	36
<b>9 Example Documentation</b>	<b>37</b>
9.1 count_primes.cpp . . . . .	37
9.2 primesieve_iterator.cpp . . . . .	37
9.3 nth_prime.cpp . . . . .	37
9.4 prev_prime.cpp . . . . .	38
9.5 store_primes_in_vector.cpp . . . . .	38
9.6 count_primes.c . . . . .	38
9.7 prev_prime.c . . . . .	38
9.8 primesieve_iterator.c . . . . .	39
9.9 nth_prime.c . . . . .	39

---

9.10 store_primes_in_array.c . . . . .	39
<b>Index</b>	<b>41</b>



# Chapter 1

## Main Page

### 1.1 About

primesieve is a C/C++ library for fast prime number generation. It generates the primes below  $10^9$  in just 0.2 seconds on a single core of an Intel Core i7-6700 3.4GHz CPU. primesieve can generate primes and prime k-tuplets up to  $2^{64}$ . primesieve's memory requirement is about  $\pi(\sqrt{n}) * 8$  bytes per thread, its run-time complexity is  $O(n \log \log n)$  operations. The recommended way to get started is to first have a look at a few C or C++ example programs. The most common use cases are iterating over primes using `next_prime()` or `prev_prime()` and storing primes in a vector or an array.

For more information please visit <https://github.com/kimwalisch/primesieve>.

### 1.2 C++ API

- [primesieve.hpp](#) - primesieve C++ header.
- [primesieve::iterator](#) - Provides the `next_prime()` and `prev_prime()` methods.
- [C++ examples](#) - Example programs that show how to use libprimesieve.

### 1.3 C API

- [primesieve.h](#) - primesieve C header.
- [primesieve\\_iterator](#) - Provides the `primesieve_next_prime()` and `primesieve_prev_prime()` functions.
- [C examples](#) - Example programs that show how to use libprimesieve.





## Chapter 2

# Namespace Index

### 2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

<a href="#">primesieve</a>	
Contains primesieve's C++ functions and classes . . . . .	<a href="#">11</a>



## Chapter 3

# Hierarchical Index

### 3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

primesieve::iterator . . . . .	17
primesieve_iterator . . . . .	21
std::runtime_error	
primesieve::primesieve_error . . . . .	20



## Chapter 4

# Class Index

### 4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<a href="#">primesieve::iterator</a>	Primesieve::iterator allows to easily iterate over primes both forwards and backwards . . . . .	17
<a href="#">primesieve::primesieve_error</a>	Primesieve throws a <a href="#">primesieve_error</a> exception if an error occurs e.g . . . . .	20
<a href="#">primesieve_iterator</a>	C prime iterator, please refer to <a href="#">iterator.h</a> for more information . . . . .	21



## Chapter 5

# File Index

### 5.1 File List

Here is a list of all documented files with brief descriptions:

<a href="#">iterator.h</a>	Primesieve_iterator allows to easily iterate over primes both forwards and backwards . . . . .	23
<a href="#">iterator.hpp</a>	The iterator class allows to easily iterate (forwards and backwards) over prime numbers . . . . .	25
<a href="#">primesieve.h</a>	Primesieve C API . . . . .	27
<a href="#">primesieve.hpp</a>	Primesieve C++ API . . . . .	33
<a href="#">primesieve_error.hpp</a>	The primesieve_error class is used for all exceptions within primesieve . . . . .	35





## Chapter 6

# Namespace Documentation

### 6.1 primesieve Namespace Reference

Contains primesieve's C++ functions and classes.

#### Classes

- class [iterator](#)  
*[primesieve::iterator](#) allows to easily iterate over primes both forwards and backwards.*
- class [primesieve\\_error](#)  
*primesieve throws a [primesieve\\_error](#) exception if an error occurs e.g.*

#### Functions

- `template<typename T >`  
`void generate\_primes (uint64_t stop, std::vector< T > *primes)`  
*Store the primes  $\leq$  stop in the primes vector.*
- `template<typename T >`  
`void generate\_primes (uint64_t start, uint64_t stop, std::vector< T > *primes)`  
*Store the primes within the interval [start, stop] in the primes vector.*
- `template<typename T >`  
`void generate\_n\_primes (uint64_t n, std::vector< T > *primes)`  
*Store the first n primes in the primes vector.*
- `template<typename T >`  
`void generate\_n\_primes (uint64_t n, uint64_t start, std::vector< T > *primes)`  
*Store the first n primes  $\geq$  start in the primes vector.*
- `uint64_t nth\_prime (int64_t n, uint64_t start=0)`  
*Find the nth prime.*
- `uint64_t count\_primes (uint64_t start, uint64_t stop)`  
*Count the primes within the interval [start, stop].*
- `uint64_t count\_twins (uint64_t start, uint64_t stop)`  
*Count the twin primes within the interval [start, stop].*
- `uint64_t count\_triplets (uint64_t start, uint64_t stop)`  
*Count the prime triplets within the interval [start, stop].*
- `uint64_t count\_quadruplets (uint64_t start, uint64_t stop)`



### 6.1.2.2 count\_quadruplets()

```
uint64_t primesieve::count_quadruplets (
    uint64_t start,
    uint64_t stop )
```

Count the prime quadruplets within the interval [start, stop].

By default all CPU cores are used, use [primesieve::set\\_num\\_threads\(int threads\)](#) to change the number of threads.

### 6.1.2.3 count\_quintuplets()

```
uint64_t primesieve::count_quintuplets (
    uint64_t start,
    uint64_t stop )
```

Count the prime quintuplets within the interval [start, stop].

By default all CPU cores are used, use [primesieve::set\\_num\\_threads\(int threads\)](#) to change the number of threads.

### 6.1.2.4 count\_sextuplets()

```
uint64_t primesieve::count_sextuplets (
    uint64_t start,
    uint64_t stop )
```

Count the prime sextuplets within the interval [start, stop].

By default all CPU cores are used, use [primesieve::set\\_num\\_threads\(int threads\)](#) to change the number of threads.

### 6.1.2.5 count\_triplets()

```
uint64_t primesieve::count_triplets (
    uint64_t start,
    uint64_t stop )
```

Count the prime triplets within the interval [start, stop].

By default all CPU cores are used, use [primesieve::set\\_num\\_threads\(int threads\)](#) to change the number of threads.

### 6.1.2.6 count\_twins()

```
uint64_t primesieve::count_twins (
    uint64_t start,
    uint64_t stop )
```

Count the twin primes within the interval [start, stop].

By default all CPU cores are used, use [primesieve::set\\_num\\_threads\(int threads\)](#) to change the number of threads.

### 6.1.2.7 `get_max_stop()`

```
uint64_t primesieve::get_max_stop ( )
```

Returns the largest valid stop number for primesieve.

#### Returns

$2^{64}-1$  (UINT64\_MAX).

### 6.1.2.8 `nth_prime()`

```
uint64_t primesieve::nth_prime (
    int64_t n,
    uint64_t start = 0 )
```

Find the nth prime.

By default all CPU cores are used, use [primesieve::set\\_num\\_threads\(int threads\)](#) to change the number of threads.

Note that each call to `nth_prime(n, start)` incurs an initialization overhead of  $O(\sqrt{\text{start}})$  even if `n` is tiny. Hence it is not a good idea to use [nth\\_prime\(\)](#) repeatedly in a loop to get the next (or previous) prime. For this use case it is better to use a [primesieve::iterator](#) which needs to be initialized only once.

#### Parameters

<i>n</i>	if <code>n = 0</code> finds the 1st prime $\geq$ start, if <code>n &gt; 0</code> finds the nth prime $>$ start, if <code>n &lt; 0</code> finds the nth prime $<$ start (backwards).
----------	---

#### Examples

[nth\\_prime.cpp](#).

### 6.1.2.9 `set_num_threads()`

```
void primesieve::set_num_threads (
    int num_threads )
```

Set the number of threads for use in `primesieve::count_*`() and [primesieve::nth\\_prime\(\)](#).

By default all CPU cores are used.

#### 6.1.2.10 `set_sieve_size()`

```
void primesieve::set_sieve_size (
    int sieve_size )
```

Set the sieve size in KiB (kibibyte).

The best sieving performance is achieved with a sieve size of your CPU's L1 or L2 cache size (per core).

##### Precondition

`sieve_size >= 8 && <= 4096.`



## Chapter 7

# Class Documentation

### 7.1 primesieve::iterator Class Reference

[primesieve::iterator](#) allows to easily iterate over primes both forwards and backwards.

```
#include <iterator.hpp>
```

#### Public Member Functions

- [iterator](#) (uint64\_t start=0, uint64\_t stop\_hint=[get\\_max\\_stop](#)())  
*Create a new iterator object.*
- [iterator](#) (const [iterator](#) &)=delete  
*[primesieve::iterator](#) objects cannot be copied.*
- [iterator](#) & **operator=** (const [iterator](#) &)=delete
- [iterator](#) ([iterator](#) &&) noexcept  
*[primesieve::iterator](#) objects support move semantics.*
- [iterator](#) & **operator=** ([iterator](#) &&) noexcept
- void [skipto](#) (uint64\_t start, uint64\_t stop\_hint=[get\\_max\\_stop](#)())  
*Reset the primesieve iterator to start.*
- uint64\_t [next\\_prime](#) ()  
*Get the next prime.*
- uint64\_t [prev\\_prime](#) ()  
*Get the previous prime.*

#### 7.1.1 Detailed Description

[primesieve::iterator](#) allows to easily iterate over primes both forwards and backwards.

Generating the first prime has a complexity of  $O(r \log \log r)$  operations with  $r = n^{0.5}$ , after that any additional prime is generated in amortized  $O(\log n \log \log n)$  operations. The memory usage is  $\text{PrimePi}(n^{0.5}) * 8$  bytes.

#### Examples

[prev\\_prime.cpp](#), and [primesieve\\_iterator.cpp](#).

## 7.1.2 Constructor & Destructor Documentation

### 7.1.2.1 iterator()

```
primesieve::iterator::iterator (
    uint64_t start = 0,
    uint64_t stop_hint = get\_max\_stop\(\) )
```

Create a new iterator object.

#### Parameters

<i>start</i>	Generate primes > start (or < start).
<i>stop_hint</i>	Stop number optimization hint, gives significant speed up if few primes are generated. E.g. if you want to generate the primes below 1000 use stop_hint = 1000.

## 7.1.3 Member Function Documentation

### 7.1.3.1 next\_prime()

```
uint64_t primesieve::iterator::next_prime ( ) [inline]
```

Get the next prime.

Returns UINT64\_MAX if next prime > 2<sup>64</sup>.

#### Examples

[primesieve\\_iterator.cpp](#).

### 7.1.3.2 prev\_prime()

```
uint64_t primesieve::iterator::prev_prime ( ) [inline]
```

Get the previous prime.

prev\_prime(n) returns 0 for n ≤ 2. Note that [next\\_prime\(\)](#) runs up to 2x faster than [prev\\_prime\(\)](#). Hence if the same algorithm can be written using either [prev\\_prime\(\)](#) or [next\\_prime\(\)](#) it is preferable to use [next\\_prime\(\)](#).

#### Examples

[prev\\_prime.cpp](#).



### 7.1.3.3 skipto()

```
void primesieve::iterator::skipto (
    uint64_t start,
    uint64_t stop_hint = get\_max\_stop\(\) )
```

Reset the primesieve iterator to start.

## Parameters

<i>start</i>	Generate primes > start (or < start).
<i>stop_hint</i>	Stop number optimization hint, gives significant speed up if few primes are generated. E.g. if you want to generate the primes below 1000 use stop_hint = 1000.

## Examples

[prev\\_prime.cpp](#), and [primesieve\\_iterator.cpp](#).

The documentation for this class was generated from the following file:

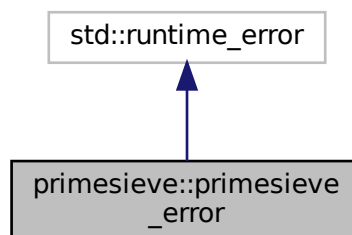
- [iterator.hpp](#)

## 7.2 primesieve::primesieve\_error Class Reference

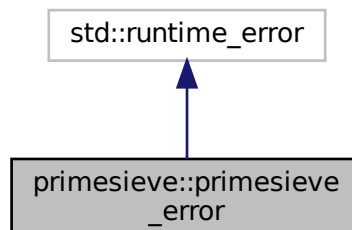
primesieve throws a [primesieve\\_error](#) exception if an error occurs e.g.

```
#include <primesieve_error.hpp>
```

Inheritance diagram for primesieve::primesieve\_error:



Collaboration diagram for primesieve::primesieve\_error:



## Public Member Functions

- **primesieve\_error** (const std::string &msg)

### 7.2.1 Detailed Description

primesieve throws a [primesieve\\_error](#) exception if an error occurs e.g.

prime > 2<sup>64</sup>.

The documentation for this class was generated from the following file:

- [primesieve\\_error.hpp](#)

## 7.3 primesieve\_iterator Struct Reference

C prime iterator, please refer to [iterator.h](#) for more information.

```
#include <iterator.h>
```

## Public Attributes

- size\_t **i**
- size\_t **last\_idx**
- uint64\_t **start**
- uint64\_t **stop**
- uint64\_t **stop\_hint**
- uint64\_t **dist**
- uint64\_t \* **primes**
- void \* **vector**
- void \* **primeGenerator**
- int **is\_error**

### 7.3.1 Detailed Description

C prime iterator, please refer to [iterator.h](#) for more information.

#### Examples

[prev\\_prime.c](#), and [primesieve\\_iterator.c](#).

The documentation for this struct was generated from the following file:

- [iterator.h](#)



## Chapter 8

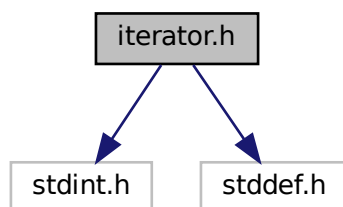
# File Documentation

### 8.1 iterator.h File Reference

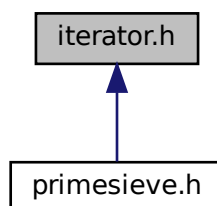
[primesieve\\_iterator](#) allows to easily iterate over primes both forwards and backwards.

```
#include <stdint.h>
#include <stddef.h>
```

Include dependency graph for iterator.h:



This graph shows which files directly or indirectly include this file:



## Classes

- struct [primesieve\\_iterator](#)  
C prime iterator, please refer to [iterator.h](#) for more information.

## Functions

- void [primesieve\\_init](#) ([primesieve\\_iterator](#) \*it)  
*Initialize the primesieve iterator before first using it.*
- void [primesieve\\_free\\_iterator](#) ([primesieve\\_iterator](#) \*it)  
*Free all memory.*
- void [primesieve\\_skipto](#) ([primesieve\\_iterator](#) \*it, uint64\_t start, uint64\_t stop\_hint)  
*Reset the primesieve iterator to start.*
- static uint64\_t [primesieve\\_next\\_prime](#) ([primesieve\\_iterator](#) \*it)  
*Get the next prime.*
- static uint64\_t [primesieve\\_prev\\_prime](#) ([primesieve\\_iterator](#) \*it)  
*Get the previous prime.*

### 8.1.1 Detailed Description

[primesieve\\_iterator](#) allows to easily iterate over primes both forwards and backwards.

Generating the first prime has a complexity of  $O(r \log \log r)$  operations with  $r = n^{0.5}$ , after that any additional prime is generated in amortized  $O(\log n \log \log n)$  operations. The memory usage is about  $\text{PrimePi}(n^{0.5}) * 8$  bytes.

The [primesieve\\_iterator.c](#) example shows how to use [primesieve\\_iterator](#). If any error occurs [primesieve\\_next\\_prime\(\)](#) and [primesieve\\_prev\\_prime\(\)](#) return `PRIMESIEVE_ERROR`. Furthermore `primesieve_iterator.is_error` is initialized to 0 and set to 1 if any error occurs.

Copyright (C) 2019 Kim Walisch, [kim.walisch@gmail.com](mailto:kim.walisch@gmail.com)

This file is distributed under the BSD License. See the COPYING file in the top level directory.

### 8.1.2 Function Documentation

#### 8.1.2.1 [primesieve\\_next\\_prime\(\)](#)

```
static uint64_t primesieve_next_prime (
    primesieve\_iterator * it ) [inline], [static]
```

Get the next prime.

Returns `UINT64_MAX` if next prime  $> 2^{64}$ .

#### Examples

[primesieve\\_iterator.c](#).

### 8.1.2.2 primesieve\_prev\_prime()

```
static uint64_t primesieve_prev_prime (
    primesieve_iterator * it ) [inline], [static]
```

Get the previous prime.

primesieve\_prev\_prime(n) returns 0 for  $n \leq 2$ . Note that [primesieve\\_next\\_prime\(\)](#) runs up to 2x faster than [primesieve\\_prev\\_prime\(\)](#). Hence if the same algorithm can be written using either [primesieve\\_prev\\_prime\(\)](#) or [primesieve\\_next\\_prime\(\)](#) it is preferable to use [primesieve\\_next\\_prime\(\)](#).

#### Examples

[prev\\_prime.c](#).

### 8.1.2.3 primesieve\_skipto()

```
void primesieve_skipto (
    primesieve_iterator * it,
    uint64_t start,
    uint64_t stop_hint )
```

Reset the primesieve iterator to start.

#### Parameters

<i>start</i>	Generate primes $>$ start (or $<$ start).
<i>stop_hint</i>	Stop number optimization hint. E.g. if you want to generate the primes below 1000 use <code>stop_hint = 1000</code> , if you don't know use <a href="#">primesieve_get_max_stop()</a> .

#### Examples

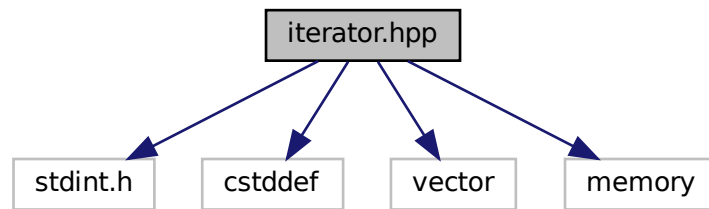
[prev\\_prime.c](#), and [primesieve\\_iterator.c](#).

## 8.2 iterator.hpp File Reference

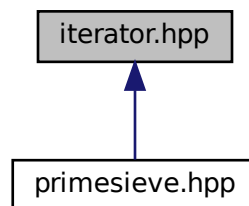
The iterator class allows to easily iterate (forwards and backwards) over prime numbers.

```
#include <stdint.h>
#include <cstdint>
#include <vector>
#include <memory>
```

Include dependency graph for iterator.hpp:



This graph shows which files directly or indirectly include this file:



## Classes

- class [primesieve::iterator](#)  
*[primesieve::iterator](#) allows to easily iterate over primes both forwards and backwards.*

## Namespaces

- [primesieve](#)  
*Contains primesieve's C++ functions and classes.*

## Functions

- `uint64_t` [primesieve::get\\_max\\_stop\(\)](#)  
*Returns the largest valid stop number for primesieve.*



### 8.2.1 Detailed Description

The iterator class allows to easily iterate (forwards and backwards) over prime numbers.

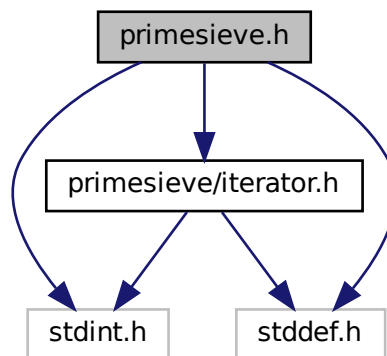
Copyright (C) 2019 Kim Walisch, [kim.walisch@gmail.com](mailto:kim.walisch@gmail.com)

This file is distributed under the BSD License. See the COPYING file in the top level directory.

## 8.3 primesieve.h File Reference

primesieve C API.

```
#include <primesieve/iterator.h>
#include <stdint.h>
#include <stddef.h>
Include dependency graph for primesieve.h:
```



### Macros

- `#define PRIMESIEVE_VERSION "7.6"`
- `#define PRIMESIEVE_VERSION_MAJOR 7`
- `#define PRIMESIEVE_VERSION_MINOR 6`
- `#define PRIMESIEVE_ERROR ((uint64_t) ~((uint64_t) 0))`  
*primesieve functions return PRIMESIEVE\_ERROR (UINT64\_MAX) if any error occurs.*

### Enumerations

- `enum {`  
`SHORT_PRIMES, USHORT_PRIMES, INT_PRIMES, UINT_PRIMES,`  
`LONG_PRIMES, ULONG_PRIMES, LONGLONG_PRIMES, ULLONG_PRIMES,`  
`INT16_PRIMES, UINT16_PRIMES, INT32_PRIMES, UINT32_PRIMES,`  
`INT64_PRIMES, UINT64_PRIMES }`

## Functions

- void \* [primesieve\\_generate\\_primes](#) (uint64\_t start, uint64\_t stop, size\_t \*size, int type)  
*Get an array with the primes inside the interval [start, stop].*
- void \* [primesieve\\_generate\\_n\\_primes](#) (uint64\_t n, uint64\_t start, int type)  
*Get an array with the first n primes  $\geq$  start.*
- uint64\_t [primesieve\\_nth\\_prime](#) (int64\_t n, uint64\_t start)  
*Find the nth prime.*
- uint64\_t [primesieve\\_count\\_primes](#) (uint64\_t start, uint64\_t stop)  
*Count the primes within the interval [start, stop].*
- uint64\_t [primesieve\\_count\\_twins](#) (uint64\_t start, uint64\_t stop)  
*Count the twin primes within the interval [start, stop].*
- uint64\_t [primesieve\\_count\\_triplets](#) (uint64\_t start, uint64\_t stop)  
*Count the prime triplets within the interval [start, stop].*
- uint64\_t [primesieve\\_count\\_quadruplets](#) (uint64\_t start, uint64\_t stop)  
*Count the prime quadruplets within the interval [start, stop].*
- uint64\_t [primesieve\\_count\\_quintuplets](#) (uint64\_t start, uint64\_t stop)  
*Count the prime quintuplets within the interval [start, stop].*
- uint64\_t [primesieve\\_count\\_sextuplets](#) (uint64\_t start, uint64\_t stop)  
*Count the prime sextuplets within the interval [start, stop].*
- void [primesieve\\_print\\_primes](#) (uint64\_t start, uint64\_t stop)  
*Print the primes within the interval [start, stop] to the standard output.*
- void [primesieve\\_print\\_twins](#) (uint64\_t start, uint64\_t stop)  
*Print the twin primes within the interval [start, stop] to the standard output.*
- void [primesieve\\_print\\_triplets](#) (uint64\_t start, uint64\_t stop)  
*Print the prime triplets within the interval [start, stop] to the standard output.*
- void [primesieve\\_print\\_quadruplets](#) (uint64\_t start, uint64\_t stop)  
*Print the prime quadruplets within the interval [start, stop] to the standard output.*
- void [primesieve\\_print\\_quintuplets](#) (uint64\_t start, uint64\_t stop)  
*Print the prime quintuplets within the interval [start, stop] to the standard output.*
- void [primesieve\\_print\\_sextuplets](#) (uint64\_t start, uint64\_t stop)  
*Print the prime sextuplets within the interval [start, stop] to the standard output.*
- uint64\_t [primesieve\\_get\\_max\\_stop](#) ()  
*Returns the largest valid stop number for primesieve.*
- int [primesieve\\_get\\_sieve\\_size](#) ()  
*Get the current set sieve size in KiB.*
- int [primesieve\\_get\\_num\\_threads](#) ()  
*Get the current set number of threads.*
- void [primesieve\\_set\\_sieve\\_size](#) (int sieve\_size)  
*Set the sieve size in KiB (kibibyte).*
- void [primesieve\\_set\\_num\\_threads](#) (int num\_threads)  
*Set the number of threads for use in [primesieve\\_count\\_\\*\(\)](#) and [primesieve\\_nth\\_prime\(\)](#).*
- void [primesieve\\_free](#) (void \*primes)  
*Deallocate a primes array created using the [primesieve\\_generate\\_primes\(\)](#) or [primesieve\\_generate\\_n\\_primes\(\)](#) functions.*
- const char \* [primesieve\\_version](#) ()  
*Get the primesieve version number, in the form "i.j"*

### 8.3.1 Detailed Description

primesieve C API.

primesieve is a library for fast prime number generation. In case an error occurs `errno` is set to `EDOM` and `PRIMESIEVE_ERROR` is returned.

Copyright (C) 2020 Kim Walisch, [kim.walisch@gmail.com](mailto:kim.walisch@gmail.com)

This file is distributed under the BSD License.

### 8.3.2 Enumeration Type Documentation

#### 8.3.2.1 anonymous enum

anonymous enum

Enumerator

SHORT_PRIMES	Generate primes of short type.
USHORT_PRIMES	Generate primes of unsigned short type.
INT_PRIMES	Generate primes of int type.
UINT_PRIMES	Generate primes of unsigned int type.
LONG_PRIMES	Generate primes of long type.
ULONG_PRIMES	Generate primes of unsigned long type.
LONGLONG_PRIMES	Generate primes of long long type.
ULONGLONG_PRIMES	Generate primes of unsigned long long type.
INT16_PRIMES	Generate primes of int16_t type.
UINT16_PRIMES	Generate primes of uint16_t type.
INT32_PRIMES	Generate primes of int32_t type.
UINT32_PRIMES	Generate primes of uint32_t type.
INT64_PRIMES	Generate primes of int64_t type.
UINT64_PRIMES	Generate primes of uint64_t type.

### 8.3.3 Function Documentation

#### 8.3.3.1 primesieve\_count\_primes()

```
uint64_t primesieve_count_primes (
    uint64_t start,
    uint64_t stop )
```

Count the primes within the interval [start, stop].

By default all CPU cores are used, use [primesieve\\_set\\_num\\_threads\(int threads\)](#) to change the number of threads.

Note that each call to [primesieve\\_count\\_primes\(\)](#) incurs an initialization overhead of  $O(\sqrt{\text{stop}})$  even if the interval [start, stop] is tiny. Hence if you have written an algorithm that makes many calls to [primesieve\\_count\\_primes\(\)](#) it may be preferable to use a [primesieve::iterator](#) which needs to be initialized only once.

#### Examples

[count\\_primes.c](#).

#### 8.3.3.2 primesieve\_count\_quadruplets()

```
uint64_t primesieve_count_quadruplets (
    uint64_t start,
    uint64_t stop )
```

Count the prime quadruplets within the interval [start, stop].

By default all CPU cores are used, use [primesieve\\_set\\_num\\_threads\(int threads\)](#) to change the number of threads.

#### 8.3.3.3 primesieve\_count\_quintuplets()

```
uint64_t primesieve_count_quintuplets (
    uint64_t start,
    uint64_t stop )
```

Count the prime quintuplets within the interval [start, stop].

By default all CPU cores are used, use [primesieve\\_set\\_num\\_threads\(int threads\)](#) to change the number of threads.

#### 8.3.3.4 primesieve\_count\_sextuplets()

```
uint64_t primesieve_count_sextuplets (
    uint64_t start,
    uint64_t stop )
```

Count the prime sextuplets within the interval [start, stop].

By default all CPU cores are used, use [primesieve\\_set\\_num\\_threads\(int threads\)](#) to change the number of threads.

### 8.3.3.5 primesieve\_count\_triplets()

```
uint64_t primesieve_count_triplets (
    uint64_t start,
    uint64_t stop )
```

Count the prime triplets within the interval [start, stop].

By default all CPU cores are used, use [primesieve\\_set\\_num\\_threads\(int threads\)](#) to change the number of threads.

### 8.3.3.6 primesieve\_count\_twins()

```
uint64_t primesieve_count_twins (
    uint64_t start,
    uint64_t stop )
```

Count the twin primes within the interval [start, stop].

By default all CPU cores are used, use [primesieve\\_set\\_num\\_threads\(int threads\)](#) to change the number of threads.

### 8.3.3.7 primesieve\_generate\_n\_primes()

```
void* primesieve_generate_n_primes (
    uint64_t n,
    uint64_t start,
    int type )
```

Get an array with the first n primes  $\geq$  start.

#### Parameters

<i>type</i>	The type of the primes to generate, e.g. INT_PRIMES.
-------------	--

#### Examples

[store\\_primes\\_in\\_array.c](#).

### 8.3.3.8 primesieve\_generate\_primes()

```
void* primesieve_generate_primes (
    uint64_t start,
    uint64_t stop,
    size_t * size,
    int type )
```

Get an array with the primes inside the interval [start, stop].

## Parameters

<i>size</i>	The size of the returned primes array.
<i>type</i>	The type of the primes to generate, e.g. INT_PRIMES.

## Examples

[store\\_primes\\_in\\_array.c](#).

**8.3.3.9 primesieve\_get\_max\_stop()**

```
uint64_t primesieve_get_max_stop ( )
```

Returns the largest valid stop number for primesieve.

## Returns

$2^{64}-1$  (UINT64\_MAX).

**8.3.3.10 primesieve\_nth\_prime()**

```
uint64_t primesieve_nth_prime (
    int64_t n,
    uint64_t start )
```

Find the nth prime.

By default all CPU cores are used, use [primesieve\\_set\\_num\\_threads\(int threads\)](#) to change the number of threads.

Note that each call to `primesieve_nth_prime(n, start)` incurs an initialization overhead of  $O(\sqrt{\text{start}})$  even if `n` is tiny. Hence it is not a good idea to use [primesieve\\_nth\\_prime\(\)](#) repeatedly in a loop to get the next (or previous) prime. For this use case it is better to use a [primesieve::iterator](#) which needs to be initialized only once.

## Parameters

<i>n</i>	if <code>n = 0</code> finds the 1st prime $\geq$ start, if <code>n &gt; 0</code> finds the nth prime $>$ start, if <code>n &lt; 0</code> finds the nth prime $<$ start (backwards).
----------	---

## Examples

[nth\\_prime.c](#).

### 8.3.3.11 primesieve\_set\_num\_threads()

```
void primesieve_set_num_threads (
    int num_threads )
```

Set the number of threads for use in `primesieve_count_*`() and `primesieve_nth_prime()`.

By default all CPU cores are used.

### 8.3.3.12 primesieve\_set\_sieve\_size()

```
void primesieve_set_sieve_size (
    int sieve_size )
```

Set the sieve size in KiB (kibibyte).

The best sieving performance is achieved with a sieve size of your CPU's L1 or L2 cache size (per core).

#### Precondition

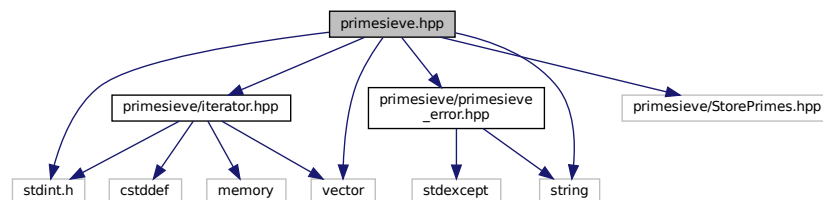
`sieve_size >= 8 && <= 4096.`

## 8.4 primesieve.hpp File Reference

primesieve C++ API.

```
#include <primesieve/iterator.hpp>
#include <primesieve/primesieve_error.hpp>
#include <primesieve/StorePrimes.hpp>
#include <stdint.h>
#include <vector>
#include <string>
```

Include dependency graph for `primesieve.hpp`:



## Namespaces

- `primesieve`

*Contains primesieve's C++ functions and classes.*

## Macros

- `#define PRIMESIEVE_VERSION "7.6"`
- `#define PRIMESIEVE_VERSION_MAJOR 7`
- `#define PRIMESIEVE_VERSION_MINOR 6`

## Functions

- `template<typename T >`  
`void primesieve::generate_primes (uint64_t stop, std::vector< T > *primes)`  
*Store the primes  $\leq$  stop in the primes vector.*
- `template<typename T >`  
`void primesieve::generate_primes (uint64_t start, uint64_t stop, std::vector< T > *primes)`  
*Store the primes within the interval [start, stop] in the primes vector.*
- `template<typename T >`  
`void primesieve::generate_n_primes (uint64_t n, std::vector< T > *primes)`  
*Store the first n primes in the primes vector.*
- `template<typename T >`  
`void primesieve::generate_n_primes (uint64_t n, uint64_t start, std::vector< T > *primes)`  
*Store the first n primes  $\geq$  start in the primes vector.*
- `uint64_t primesieve::nth_prime (int64_t n, uint64_t start=0)`  
*Find the nth prime.*
- `uint64_t primesieve::count_primes (uint64_t start, uint64_t stop)`  
*Count the primes within the interval [start, stop].*
- `uint64_t primesieve::count_twins (uint64_t start, uint64_t stop)`  
*Count the twin primes within the interval [start, stop].*
- `uint64_t primesieve::count_triplets (uint64_t start, uint64_t stop)`  
*Count the prime triplets within the interval [start, stop].*
- `uint64_t primesieve::count_quadruplets (uint64_t start, uint64_t stop)`  
*Count the prime quadruplets within the interval [start, stop].*
- `uint64_t primesieve::count_quintuplets (uint64_t start, uint64_t stop)`  
*Count the prime quintuplets within the interval [start, stop].*
- `uint64_t primesieve::count_sextuplets (uint64_t start, uint64_t stop)`  
*Count the prime sextuplets within the interval [start, stop].*
- `void primesieve::print_primes (uint64_t start, uint64_t stop)`  
*Print the primes within the interval [start, stop] to the standard output.*
- `void primesieve::print_twins (uint64_t start, uint64_t stop)`  
*Print the twin primes within the interval [start, stop] to the standard output.*
- `void primesieve::print_triplets (uint64_t start, uint64_t stop)`  
*Print the prime triplets within the interval [start, stop] to the standard output.*
- `void primesieve::print_quadruplets (uint64_t start, uint64_t stop)`  
*Print the prime quadruplets within the interval [start, stop] to the standard output.*
- `void primesieve::print_quintuplets (uint64_t start, uint64_t stop)`  
*Print the prime quintuplets within the interval [start, stop] to the standard output.*
- `void primesieve::print_sextuplets (uint64_t start, uint64_t stop)`  
*Print the prime sextuplets within the interval [start, stop] to the standard output.*
- `uint64_t primesieve::get_max_stop ()`  
*Returns the largest valid stop number for primesieve.*
- `int primesieve::get_sieve_size ()`  
*Get the current set sieve size in KiB.*
- `int primesieve::get_num_threads ()`



- *Get the current set number of threads.*  
void `primesieve::set_sieve_size` (int sieve\_size)
- *Set the sieve size in KiB (kibibyte).*  
void `primesieve::set_num_threads` (int num\_threads)
- *Set the number of threads for use in `primesieve::count_*`() and `primesieve::nth_prime`().*  
std::string `primesieve::primesieve_version` ()
- *Get the primesieve version number, in the form "i.j".*

### 8.4.1 Detailed Description

primesieve C++ API.

primesieve is a library for fast prime number generation, in case an error occurs a `primesieve::primesieve_error` exception (derived from `std::runtime_error`) is thrown.

Copyright (C) 2020 Kim Walisch, [kim.walisch@gmail.com](mailto:kim.walisch@gmail.com)

This file is distributed under the BSD License.

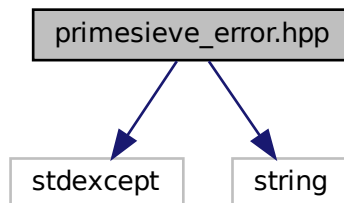
## 8.5 primesieve\_error.hpp File Reference

The `primesieve_error` class is used for all exceptions within primesieve.

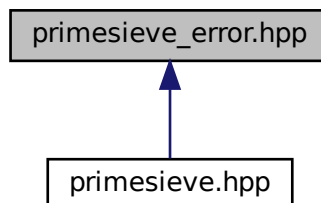
```
#include <stdexcept>
```

```
#include <string>
```

Include dependency graph for `primesieve_error.hpp`:



This graph shows which files directly or indirectly include this file:



## Classes

- class [primesieve::primesieve\\_error](#)  
*primesieve throws a [primesieve\\_error](#) exception if an error occurs e.g.*

## Namespaces

- [primesieve](#)  
*Contains primesieve's C++ functions and classes.*

### 8.5.1 Detailed Description

The `primesieve_error` class is used for all exceptions within `primesieve`.

Copyright (C) 2017 Kim Walisch, [kim.walisch@gmail.com](mailto:kim.walisch@gmail.com)

This file is distributed under the BSD License. See the `COPYING` file in the top level directory.

## Chapter 9

# Example Documentation

### 9.1 count\_primes.cpp

This example shows how to count primes.

```
#include <primesieve.hpp>
#include <stdint.h>
#include <iostream>
int main()
{
    uint64_t count = primesieve::count_primes(0, 1000);
    std::cout << "Primes below 1000 = " << count << std::endl;
    return 0;
}
```

### 9.2 primesieve\_iterator.cpp

Iterate over primes using `primesieve::iterator`.

```
#include <primesieve.hpp>
#include <iostream>
int main()
{
    primesieve::iterator it;
    uint64_t prime = it.next_prime();
    uint64_t sum = 0;
    // iterate over the primes below 10^9
    for (; prime < 1000000000ull; prime = it.next_prime())
        sum += prime;
    std::cout << "Sum of the primes below 10^9 = " << sum << std::endl;
    // generate primes > 1000
    it.skipto(1000);
    prime = it.next_prime();
    for (; prime < 1100; prime = it.next_prime())
        std::cout << prime << std::endl;
    return 0;
}
```

### 9.3 nth\_prime.cpp

Find the *n*th prime.

```
#include <primesieve.hpp>
#include <stdint.h>
#include <iostream>
#include <cstdlib>
int main(int, char** argv)
{
    uint64_t n = 1000;
    if (argv[1])
        n = std::atol(argv[1]);
    uint64_t nth_prime = primesieve::nth_prime(n);
    std::cout << n << "th prime = " << nth_prime << std::endl;
    return 0;
}
```

## 9.4 prev\_prime.cpp

Iterate backwards over primes using `primesieve::iterator`.

```
#include <primesieve.hpp>
#include <iostream>
int main()
{
    primesieve::iterator it;
    it.skipto(2000);
    uint64_t prime = it.prev_prime();
    // iterate over primes from 2000 to 1000
    for (; prime >= 1000; prime = it.prev_prime())
        std::cout << prime << std::endl;
    return 0;
}
```

## 9.5 store\_primes\_in\_vector.cpp

Store primes in a `std::vector` using `primesieve`.

```
#include <primesieve.hpp>
#include <vector>
int main()
{
    std::vector<int> primes;
    // Store primes <= 1000
    primesieve::generate_primes(1000, &primes);
    primes.clear();
    // Store primes inside [1000, 2000]
    primesieve::generate_primes(1000, 2000, &primes);
    primes.clear();
    // Store first 1000 primes
    primesieve::generate_n_primes(1000, &primes);
    primes.clear();
    // Store first 10 primes >= 1000
    primesieve::generate_n_primes(10, 1000, &primes);
    return 0;
}
```

## 9.6 count\_primes.c

C program that shows how to count primes.

```
#include <primesieve.h>
#include <inttypes.h>
#include <stdio.h>
int main()
{
    uint64_t count = primesieve_count_primes(0, 1000);
    printf("Primes below 1000 = %" PRIu64 "\n", count);
    return 0;
}
```

## 9.7 prev\_prime.c

Iterate backwards over primes using `primesieve_iterator`. Note that `primesieve_next_prime()` runs up to 2x faster and uses only half as much memory as `primesieve_prev_prime()`. Hence if it is possible to write the same algorithm using either `primesieve_prev_prime()` or `primesieve_next_prime()` then it is preferable to use `primesieve_next_prime()`.

```
#include <primesieve.h>
#include <inttypes.h>
#include <stdio.h>
int main()
{
    primesieve_iterator it;
    primesieve_init(&it);
    /* primesieve_skipto(&it, start_number, stop_hint) */
    primesieve_skipto(&it, 2000, 1000);
    uint64_t prime;
    /* iterate over primes from 2000 to 1000 */
    while ((prime = primesieve_prev_prime(&it)) >= 1000)
        printf("%" PRIu64 "\n", prime);
    primesieve_free_iterator(&it);
    return 0;
}
```

## 9.8 primesieve\_iterator.c

Iterate over primes using C [primesieve\\_iterator](#).

```
#include <primesieve.h>
#include <inttypes.h>
#include <stdio.h>
int main()
{
    primesieve_iterator it;
    primesieve_init(&it);
    uint64_t sum = 0;
    uint64_t prime = 0;
    /* iterate over the primes below 10^9 */
    while ((prime = primesieve_next_prime(&it)) < 1000000000ull)
        sum += prime;
    printf("Sum of the primes below 10^9 = %" PRIu64 "\n", sum);
    /* generate primes > 1000 */
    primesieve_skipto(&it, 1000, 1100);
    while ((prime = primesieve_next_prime(&it)) < 1100)
        printf("%" PRIu64 "\n", prime);
    primesieve_free_iterator(&it);
    return 0;
}
```

## 9.9 nth\_prime.c

C program that finds the nth prime.

```
#include <primesieve.h>
#include <stdlib.h>
#include <inttypes.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    uint64_t n = 1000;
    if (argc > 1 && argv[1])
        n = atol(argv[1]);
    uint64_t prime = primesieve_nth_prime(n, 0);
    printf("%" PRIu64 "th prime = %" PRIu64 "\n", n, prime);
    return 0;
}
```

## 9.10 store\_primes\_in\_array.c

Store primes in a C array.

```
#include <primesieve.h>
#include <stdio.h>
int main()
{
    uint64_t start = 0;
    uint64_t stop = 1000;
    size_t i;
    size_t size;
    /* store the primes below 1000 */
    int* primes = (int*) primesieve_generate_primes(start, stop, &size, INT_PRIMES);
    for (i = 0; i < size; i++)
        printf("%i\n", primes[i]);
    primesieve_free(primes);
    uint64_t n = 1000;
    /* store the first 1000 primes */
    primes = (int*) primesieve_generate_n_primes(n, start, INT_PRIMES);
    for (i = 0; i < n; i++)
        printf("%i\n", primes[i]);
    primesieve_free(primes);
    return 0;
}
```



# Index

- count\_primes
  - primesieve, [12](#)
- count\_quadruplets
  - primesieve, [12](#)
- count\_quintuplets
  - primesieve, [13](#)
- count\_sextuplets
  - primesieve, [13](#)
- count\_triplets
  - primesieve, [13](#)
- count\_twins
  - primesieve, [13](#)
- get\_max\_stop
  - primesieve, [13](#)
- INT16\_PRIMES
  - primesieve.h, [29](#)
- INT32\_PRIMES
  - primesieve.h, [29](#)
- INT64\_PRIMES
  - primesieve.h, [29](#)
- INT\_PRIMES
  - primesieve.h, [29](#)
- iterator
  - primesieve::iterator, [18](#)
- iterator.h, [23](#)
  - primesieve\_next\_prime, [24](#)
  - primesieve\_prev\_prime, [24](#)
  - primesieve\_skipto, [25](#)
- iterator.hpp, [25](#)
- LONG\_PRIMES
  - primesieve.h, [29](#)
- LONGLONG\_PRIMES
  - primesieve.h, [29](#)
- next\_prime
  - primesieve::iterator, [18](#)
- nth\_prime
  - primesieve, [14](#)
- prev\_prime
  - primesieve::iterator, [18](#)
- primesieve, [11](#)
  - count\_primes, [12](#)
  - count\_quadruplets, [12](#)
  - count\_quintuplets, [13](#)
  - count\_sextuplets, [13](#)
  - count\_triplets, [13](#)
  - count\_twins, [13](#)
  - get\_max\_stop, [13](#)
  - nth\_prime, [14](#)
  - set\_num\_threads, [14](#)
  - set\_sieve\_size, [14](#)
- primesieve.h, [27](#)
  - INT16\_PRIMES, [29](#)
  - INT32\_PRIMES, [29](#)
  - INT64\_PRIMES, [29](#)
  - INT\_PRIMES, [29](#)
  - LONG\_PRIMES, [29](#)
  - LONGLONG\_PRIMES, [29](#)
  - primesieve\_count\_primes, [29](#)
  - primesieve\_count\_quadruplets, [30](#)
  - primesieve\_count\_quintuplets, [30](#)
  - primesieve\_count\_sextuplets, [30](#)
  - primesieve\_count\_triplets, [30](#)
  - primesieve\_count\_twins, [31](#)
  - primesieve\_generate\_n\_primes, [31](#)
  - primesieve\_generate\_primes, [31](#)
  - primesieve\_get\_max\_stop, [32](#)
  - primesieve\_nth\_prime, [32](#)
  - primesieve\_set\_num\_threads, [32](#)
  - primesieve\_set\_sieve\_size, [33](#)
  - SHORT\_PRIMES, [29](#)
  - UINT16\_PRIMES, [29](#)
  - UINT32\_PRIMES, [29](#)
  - UINT64\_PRIMES, [29](#)
  - UINT\_PRIMES, [29](#)
  - ULONG\_PRIMES, [29](#)
  - ULONGLONG\_PRIMES, [29](#)
  - USHORT\_PRIMES, [29](#)
- primesieve.hpp, [33](#)
- primesieve::iterator, [17](#)
  - iterator, [18](#)
  - next\_prime, [18](#)
  - prev\_prime, [18](#)
  - skipto, [18](#)
- primesieve::primesieve\_error, [20](#)
- primesieve\_count\_primes
  - primesieve.h, [29](#)
- primesieve\_count\_quadruplets
  - primesieve.h, [30](#)
- primesieve\_count\_quintuplets
  - primesieve.h, [30](#)
- primesieve\_count\_sextuplets
  - primesieve.h, [30](#)
- primesieve\_count\_triplets
  - primesieve.h, [30](#)
- primesieve\_count\_twins

- primesieve.h, [31](#)
- primesieve\_error.hpp, [35](#)
- primesieve\_generate\_n\_primes
  - primesieve.h, [31](#)
- primesieve\_generate\_primes
  - primesieve.h, [31](#)
- primesieve\_get\_max\_stop
  - primesieve.h, [32](#)
- primesieve\_iterator, [21](#)
- primesieve\_next\_prime
  - iterator.h, [24](#)
- primesieve\_nth\_prime
  - primesieve.h, [32](#)
- primesieve\_prev\_prime
  - iterator.h, [24](#)
- primesieve\_set\_num\_threads
  - primesieve.h, [32](#)
- primesieve\_set\_sieve\_size
  - primesieve.h, [33](#)
- primesieve\_skipto
  - iterator.h, [25](#)
- set\_num\_threads
  - primesieve, [14](#)
- set\_sieve\_size
  - primesieve, [14](#)
- SHORT\_PRIMES
  - primesieve.h, [29](#)
- skipto
  - primesieve::iterator, [18](#)
- UINT16\_PRIMES
  - primesieve.h, [29](#)
- UINT32\_PRIMES
  - primesieve.h, [29](#)
- UINT64\_PRIMES
  - primesieve.h, [29](#)
- UINT\_PRIMES
  - primesieve.h, [29](#)
- ULONG\_PRIMES
  - primesieve.h, [29](#)
- ULONGLONG\_PRIMES
  - primesieve.h, [29](#)
- USHORT\_PRIMES
  - primesieve.h, [29](#)